Fred2 Documentation

Author

Nov 13, 2018

Contents

Welcome to the class and function documentation of FRED2.

Tutorials on how to use FRED2 can be found at:

https://github.com/FRED-2/Fred2/tree/master/Fred2/tutorials.

CHAPTER 1

Basic

1.1 Fred2.Core Module

1.1.1 Core.Allele

class Fred2.Core.Allele.**Allele**(*name*, *prob=None*)
Bases: *Fred2.Core.Base.MetadataLogger*

This class represents an HLA Allele and stores additional information

get_metadata(*label*, *only_first=False*)
Getter for the saved metadata with the key `label`

Parameters

- **label** (*str*) – key for the metadata that is inferred
- **only_first** (*bool*) – true if only the the first element of the matadata list is to be returned

log_metadata(*label*, *value*)
Inserts a new metadata

Parameters

- **label** (*str*) – key for the metadata that will be added
- **value** (*list(object)*) – any kindy of additional value that should be kept

class Fred2.Core.Allele.**AlleleFactory**
Bases: `type`

mro() → list
return a type's method resolution order

class Fred2.Core.Allele.**CombinedAllele**(*name*, *prob=None*)
Bases: *Fred2.Core.Allele.Allele*

This class represents combined HLA class II Alleles with an alpha and beta chain

get_metadata(*label*, *only_first=False*)
Getter for the saved metadata with the key `label`

Parameters

- **label** (*str*) – key for the metadata that is inferred
- **only_first** (*bool*) – true if only the the first element of the matadata list is to be returned

locus

log_metadata(*label*, *value*)
Inserts a new metadata

Parameters

- **label** (*str*) – key for the metadata that will be added
- **value** (*list(object)*) – any kindy of additional value that should be kept

subtype

supertype

class Fred2.Core.Allele.**MouseAllele**(*name*, *prob=None*)
Bases: *Fred2.Core.Allele.Allele*

This class represents a mouse MHC allele with the following nomenclature: H2-Xxx

http://www.imgt.org/IMGTrepertoireMHC/LocusGenes/index.php?repertoire=listIG_TR&species=mouse&group=MHC

get_metadata(*label*, *only_first=False*)
Getter for the saved metadata with the key `label`

Parameters

- **label** (*str*) – key for the metadata that is inferred
- **only_first** (*bool*) – true if only the the first element of the matadata list is to be returned

locus

log_metadata(*label*, *value*)
Inserts a new metadata

Parameters

- **label** (*str*) – key for the metadata that will be added
- **value** (*list(object)*) – any kindy of additional value that should be kept

subtype

supertype

1.1.2 Core.Base

https://docs.python.org/3/library/abc.html

class Fred2.Core.Base.**ACleavageFragmentPrediction**
Bases: object

cleavagePos
Parameter specifying the position of aa (within the prediction window) after which the sequence is cleaved

name
The name of the predictor

predict(*aa_seq*, ***kwargs*)
Predicts the probability that the fragment can be produced by the proteasom

Parameters **aa_seq** (*Peptide*) – The sequence to be cleaved

Returns Returns a *AResult* object for the specified Bio.Seq

Return type *AResult*

supportedLength
The supported lengths of the predictor

version
Parameter specifying the version of the prediction method

class Fred2.Core.Base.**ACleavageSitePrediction**
Bases: object

cleavagePos
Parameter specifying the position of aa (within the prediction window) after which the sequence is cleaved (starting from 1)

name
The name of the predictor

predict(*aa_seq*, ***kwargs*)
Predicts the proteasomal cleavage site of the given sequences

Parameters **aa_seq** (*Peptide* or *Protein*) – The sequence to be cleaved

Returns Returns a *AResult* object for the specified Bio.Seq

Return type *AResult*

supportedLength
The supported lengths of the predictor

version
Parameter specifying the version of the prediction method

class Fred2.Core.Base.**AEpitopePrediction**
Bases: object

convert_alleles(*alleles*)
Converts alleles into the internal allele representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The alleles for which the internal predictor representation is needed

Returns Returns a string representation of the input alleles

Return type list(str)

name
The name of the predictor

predict(*peptides*, *alleles=None*, ***kwargs*)
Predicts the binding affinity for a given peptide or peptide lists for a given list of alleles. If alleles is not

given, predictions for all valid alleles of the predictor is performed. If, however, a list of alleles is given, predictions for the valid allele subset is performed.

Parameters

- **peptides** (*Peptide* or list(*Peptide*)) – The peptide objects for which predictions should be performed
- **alleles** (*Allele*/list(*Allele*)) – An *Allele* or list of *Allele* for which prediction models should be used

Returns Returns a *AResult* object for the specified *Peptide* and *Allele*

Return type *AResult*

supportedAlleles
A list of valid allele models

supportedLength
A list of supported peptide lengths

version
The version of the predictor

class Fred2.Core.Base.**AExternal**
Bases: object

Base class for external tools

command
Defines the commandline call for external tool

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) –

- Optional specification of executable path if deviant from self.__command

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

parse_external_result(*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

class Fred2.Core.Base.**AHLATyping**
Bases: object

name
The name of the predictor

predict(*ngsFile*, *output*, ***kwargs*)
Prediction method for inferring the HLA typing

Parameters

- **ngsFile** (*str*) – The path to the input file containing the NGS reads
- **output** (*str*) – The path to the output file or directory

Returns A list of HLA alleles representing the genotype predicted by the algorithm

Return type list(*Allele*)

version
Parameter specifying the version of the prediction method

class Fred2.Core.Base.**APluginRegister**(*name*, *bases*, *nmspc*)
Bases: abc.ABCMeta

This class allows automatic registration of new plugins.

mro() → list
return a type's method resolution order

register(*subclass*)
Register a virtual subclass of an ABC.

class Fred2.Core.Base.**ASVM**
Bases: object

Base class for SVM prediction tools

encode(*peptides*)
Returns the feature encoding for peptides

Parameters **peptides** (list(*Peptide*)/*Peptide*) – List of or a single *Peptide* object

Returns Feature encoding of the Peptide objects

Return type list(Object)

class Fred2.Core.Base.**ATAPPrediction**
Bases: object

name
The name of the predictor

predict(*peptides*, ***kwargs*)
Predicts the TAP affinity for the given sequences

Parameters **peptides** (list(*Peptide*)/*Peptide*) – *Peptide* for which TAP affinity should be predicted

Returns Returns a TAPResult object

Return type TAPResult

supportedLength
The supported lengths of the predictor

version
Parameter specifying the version of the prediction method

class Fred2.Core.Base.**MetadataLogger**
Bases: object

This class provides a simple interface for assigning additional metadata to any object in our data model. Examples: storing ANNOVAR columns like depth, base count, dbSNP id, quality information for variants, additional prediction information for peptides etc. This functionality is not used from core methods of FRED2.

The saved values are accessed via `log_metadata()` and `get_metadata()`

get_metadata(*label*, *only_first=False*)
Getter for the saved metadata with the key `label`

Parameters

- **label** (*str*) – key for the metadata that is inferred
- **only_first** (*bool*) – true if only the the first element of the matadata list is to be returned

log_metadata(*label*, *value*)
Inserts a new metadata

Parameters

- **label** (*str*) – key for the metadata that will be added
- **value** (*list(object)*) – any kindy of additional value that should be kept

Fred2.Core.Base.**deprecated**(*func*)
This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

1.1.3 Core.Generator

Fred2.Core.Generator.**generate_peptides_from_proteins**(*proteins*, *window_size*, *peptides=None*)
Creates all *Peptide* for a given window size, from a given *Protein*.

The result is a generator.

Parameters

- **proteins** (list(*Protein*) or *Protein*) – (Iterable of) protein(s) from which a list of unique peptides should be generated
- **window_size** (*int*) – Size of peptide fragments
- **peptides** (list(*Peptide*)) – A list of peptides to update during peptide generation (usa case: Adding and updating Peptides of newly generated Proteins)

Returns A unique generator of peptides

Return type Generator(*Peptide*)

Fred2.Core.Generator.**generate_peptides_from_variants**(*vars*, *length*, *dbadapter*, *id_type*, *peptides=None*, *table='Standard'*, *stop_symbol='*'*, *to_stop=True*, *cds=False*, *db='hsapiens_gene_ensembl'*)
Generates *Peptide* from *Variant* and avoids the construction of all possible combinations of heterozygous variants by considering only those within the peptide sequence window. This reduces the number of combinations from 2^m with m = #Heterozygous Variants to 2^k with k<<m and k = #Heterozygous Variants within peptide window (and all frame-shift mutations that occurred prior to the current peptide window).

The result is a generator.

Parameters

- **vars** (list(*Variant*)) – A list of variant objects to construct peptides from
- **length** (*int*) – The length of the peptides to construct
- **dbadapter** (ADBAdapter) – A ADBAdapter to extract relevant transcript information
- **id_type** (EIdentifierTypes()) – The type of the transcript IDs used in annotation of variants (e.g. REFSEQ, ENSAMBLE)
- **peptides** (list(*Peptide*)) – A list of pre existing peptides that should be updated
- **table** (*str*) – Which codon table to use? This can be either a name (string), an NCBI identifier (integer), or a CodonTable object (useful for non-standard genetic codes). Defaults to the 'Standard' table
- **stop_symbol** (*str*) – Single character string, what to use for any terminators, defaults to the asterisk, '*'
- **to_stop** (*bool*) – Boolean, defaults to False meaning do a full translation continuing on past any stop codons (translated as the specified stop_symbol). If True, translation is terminated at the first in frame stop codon (and the stop_symbol is not appended to the returned protein sequence)
- **cds** (*bool*) – cds - Boolean, indicates this is a complete CDS. If True, this checks the sequence starts with a valid alternative start codon (which will be translated as methionine, M), that the sequence length is a multiple of three, and that there is a single in frame stop codon at the end (this will be excluded from the protein sequence, regardless of the to_stop option). If these tests fail, an exception is raised

Returns A list of unique (polymorphic) peptides

Return type Generator(*Peptide*)

Raises

- **ValueError** – If incorrect table argument is pasted
- **TranslationError** – If sequence is not multiple of three, or first codon is not a start codon, or last codon is not a stop codon, or an extra stop codon was found in frame, or codon is non-valid

Fred2.Core.Generator.**generate_proteins_from_transcripts**(*transcripts*, *table='Standard'*, *stop_symbol='*'*, *to_stop=True*, *cds=False*)

Enables the translation from a *Transcript* to a *Protein* instance. The result is a generator.

The result is a generator.

Parameters

- **transcripts** (list(*Transcript*) or *Transcript*) – A list of or a single transcripts to translate
- **table** (*str*) – Which codon table to use? This can be either a name (string), an NCBI identifier (integer), or a CodonTable object (useful for non-standard genetic codes). Defaults to the 'Standard' table
- **stop_symbol** (*str*) – Single character string, what to use for any terminators, defaults to the asterisk, '*'

- **to_stop** (*bool*) – Translates sequence and passes any stop codons if False (default True)(translated as the specified stop_symbol). If True, translation is terminated at the first in frame stop codon (and the stop_symbol is not appended to the returned protein sequence)
- **cds** (*bool*) – Boolean, indicates this is a complete CDS. If True, this checks the sequence starts with a valid alternative start codon (which will be translated as methionine, M), that the sequence length is a multiple of three, and that there is a single in frame stop codon at the end (this will be excluded from the protein sequence, regardless of the to_stop option). If these tests fail, an exception is raised

Returns The protein that corresponds to the transcript

Return type Generator(*Protein*)

Raises

- **ValueError** – If incorrect table argument is pasted
- **TranslationError** – If sequence is not multiple of three, or first codon is not a start codon, or last codon ist not a stop codon, or an extra stop codon was found in frame, or codon is non-valid

Fred2.Core.Generator.**generate_transcripts_from_variants**(*vars*, *dbadapter*, *id_type*, *db='hsapiens_gene_ensembl'*)

Generates all possible transcript *Transcript* based on the given *Variant*.

The result is a generator.

Parameters

- **vars** (list(*Variant*)) – A list of variants for which transcripts should be build
- **id_type** (EIdentifierTypes()) – The type of the transcript IDs used in annotation of variants (e.g. REFSEQ, ENSAMBLE)

Param dbadapter: a DBAdapter to fetch the transcript sequences

Returns A generator of transcripts with all possible variations determined by the given variant list

Return type Generator(:class:'~Fred2.Core.Transcript.Transcript)

Invariant Variants are considered to be annotated from forward strand, regardless of the transcripts real orientation

1.1.4 Core.Peptide

class Fred2.Core.Peptide.**Peptide**(*seq*, *protein_pos=None*)

Bases: *Fred2.Core.Base.MetadataLogger*, Bio.Seq.Seq

This class encapsulates a *Peptide*, belonging to one or several *Protein*.

Note: For accessing and manipulating the sequence see also Bio.Seq.Seq (from Biopython)

---

back_transcribe()

Return the DNA sequence from an RNA sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG",
...                     IUPAC.unambiguous_rna)
```

(continues on next page)

(continued from previous page)

```
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
>>> messenger_rna.back_transcribe()
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
```

Trying to back-transcribe a protein or DNA sequence raises an exception:

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.back_transcribe()
Traceback (most recent call last):
  ...
ValueError: Proteins cannot be back transcribed!
```

complement()

Return the complement sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_dna = Seq("CCCCCGATAG", IUPAC.unambiguous_dna)
>>> my_dna
Seq('CCCCCGATAG', IUPACUnambiguousDNA())
>>> my_dna.complement()
Seq('GGGGGCTATC', IUPACUnambiguousDNA())
```

You can of course used mixed case sequences,

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("CCCCCgatA-GD", generic_dna)
>>> my_dna
Seq('CCCCCgatA-GD', DNAAlphabet())
>>> my_dna.complement()
Seq('GGGGGctaT-CH', DNAAlphabet())
```

Note in the above example, ambiguous character D denotes G, A or T so its complement is H (for C, T or A).

Trying to complement a protein sequence raises an exception.

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.complement()
Traceback (most recent call last):
  ...
ValueError: Proteins do not have complements!
```

count(*sub*, *start=0*, *end=9223372036854775807*)

Return a non-overlapping count, like that of a python string.

This behaves like the python string method of the same name, which does a non-overlapping count!

For an overlapping search use the newer count_overlap() method.

Returns an integer, the number of occurrences of substring argument sub in the (sub)sequence given by [start:end]. Optional arguments start and end are interpreted as in slice notation.

Arguments:

- sub - a string or another Seq object to look for

- start - optional integer, slice start
- end - optional integer, slice end

e.g.

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AAAATGA")
>>> print(my_seq.count("A"))
5
>>> print(my_seq.count("ATG"))
1
>>> print(my_seq.count(Seq("AT")))
1
>>> print(my_seq.count("AT", 2, -1))
1
```

HOWEVER, please note because python strings and Seq objects (and MutableSeq objects) do a non-overlapping search, this may not give the answer you expect:

```
>>> "AAAA".count("AA")
2
>>> print(Seq("AAAA").count("AA"))
2
```

An overlapping search, as implemented in .count_overlap(), would give the answer as three!

count_overlap(*sub*, *start=0*, *end=9223372036854775807*)

Return an overlapping count.

For a non-overlapping search use the count() method.

Returns an integer, the number of occurrences of substring argument sub in the (sub)sequence given by [start:end]. Optional arguments start and end are interpreted as in slice notation.

Arguments:

- sub - a string or another Seq object to look for
- start - optional integer, slice start
- end - optional integer, slice end

e.g.

```
>>> from Bio.Seq import Seq
>>> print(Seq("AAAA").count_overlap("AA"))
3
>>> print(Seq("ATATATATA").count_overlap("ATA"))
4
>>> print(Seq("ATATATATA").count_overlap("ATA", 3, -1))
1
```

Where substrings do not overlap, should behave the same as the count() method:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AAAATGA")
>>> print(my_seq.count_overlap("A"))
5
>>> my_seq.count_overlap("A") == my_seq.count("A")
True
```

(continues on next page)

(continued from previous page)

```
>>> print(my_seq.count_overlap("ATG"))
1
>>> my_seq.count_overlap("ATG") == my_seq.count("ATG")
True
>>> print(my_seq.count_overlap(Seq("AT")))
1
>>> my_seq.count_overlap(Seq("AT")) == my_seq.count(Seq("AT"))
True
>>> print(my_seq.count_overlap("AT", 2, -1))
1
>>> my_seq.count_overlap("AT", 2, -1) == my_seq.count("AT", 2, -1)
True
```

HOWEVER, do not use this method for such cases because the count() method is much for efficient.

endswith(*suffix*, *start=0*, *end=9223372036854775807*)

Return True if the Seq ends with the given suffix, False otherwise.

This behaves like the python string method of the same name.

Return True if the sequence ends with the specified suffix (a string or another Seq object), False otherwise. With optional start, test sequence beginning at that position. With optional end, stop comparing sequence at that position. suffix can also be a tuple of strings to try. e.g.

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.endswith("UUG")
True
>>> my_rna.endswith("AUG")
False
>>> my_rna.endswith("AUG", 0, 18)
True
>>> my_rna.endswith(("UCC", "UCA", "UUG"))
True
```

find(*sub*, *start=0*, *end=9223372036854775807*)

Find method, like that of a python string.

This behaves like the python string method of the same name.

Returns an integer, the index of the first occurrence of substring argument sub in the (sub)sequence given by [start:end].

Arguments:

- sub - a string or another Seq object to look for
- start - optional integer, slice start
- end - optional integer, slice end

Returns -1 if the subsequence is NOT found.

e.g. Locating the first typical start codon, AUG, in an RNA sequence:

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.find("AUG")
3
```

get_all_proteins()
Returns all *Protein* objects associated with the *Peptide*

Returns A list of *Protein*

Return type list(*Protein*)

get_all_transcripts()
Returns a list of *Transcript* objects that are associated with the *Peptide*

Returns A list of *Transcript*

Return type list(*Transcript*)

get_metadata(*label*, *only_first=False*)
Getter for the saved metadata with the key `label`

Parameters

- **label** (*str*) – key for the metadata that is inferred
- **only_first** (*bool*) – true if only the the first element of the matadata list is to be returned

get_protein(*transcript_id*)
Returns a specific protein object identified by a unique transcript-ID

Parameters **transcript_id** (*str*) – A *Transcript* ID

Returns A *Protein*

Return type *Protein*

get_protein_positions(*transcript_id*)
Returns all positions of origin for a given *Protein* identified by its transcript-ID

Parameters **transcript_id** (*str*) – The unique transcript ID of the *Protein* in question

Returns A list of positions within the protein from which the *Peptide* originated (starts at 0)

Return type list(int)

get_transcript(*transcript_id*)
Returns a specific *Transcript* object identified by a unique transcript-ID

Parameters **transcript_id** (*str*) – A *Transcript* ID

Returns A *Transcript*

Return type *Transcript*

get_variants_by_protein(*transcript_id*)
Returns all *Variant* of a *Protein* that have influenced the *Peptide* sequence

Parameters **transcript_id** (*str*) – *Transcript* ID of the specific protein in question

Returns A list variants that influenced the peptide sequence

Return type list(*Variant*)

Raises **KeyError** – If peptide does not originate from specified *Protein*

get_variants_by_protein_position(*transcript_id*, *protein_pos*)
Returns all *Variant* and their relative position to the peptide sequence of a given *Protein* and protein position

Parameters

- **transcript_id** (*str*) – A *Transcript* ID of the specific protein in question
- **protein_pos** (*int*) – The *Protein* position at which the peptides sequence starts in the protein

Returns Dictionary of relative position of variants in peptide (starts at 0) and associated variants that influenced the peptide sequence

Return type dict(int,list(*Variant*))

Raises

ValueError If *Peptide* does not start at specified position

KeyError If *Peptide* does not originate from specified *Protein*

log_metadata(*label*, *value*)
Inserts a new metadata

Parameters

- **label** (*str*) – key for the metadata that will be added
- **value** (*list(object)*) – any kindy of additional value that should be kept

lower()
Return a lower case copy of the sequence.

This will adjust the alphabet if required. Note that the IUPAC alphabets are upper case only, and thus a generic alphabet must be substituted.

```
>>> from Bio.Alphabet import Gapped, generic_dna
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
>>> my_seq = Seq("CGGTACGCTTATGTCACGTAG*AAAAAA",
...              Gapped(IUPAC.unambiguous_dna, "*"))
>>> my_seq
Seq('CGGTACGCTTATGTCACGTAG*AAAAAA', Gapped(IUPACUnambiguousDNA(), '*'))
>>> my_seq.lower()
Seq('cggtacgcttatgtcacgtag*aaaaaa', Gapped(DNAAlphabet(), '*'))
```

See also the upper method.

lstrip(*chars=None*)
Return a new Seq object with leading (left) end stripped.

This behaves like the python string method of the same name.

Optional argument chars defines which characters to remove. If omitted or None (default) then as for the python string method, this defaults to removing any white space.

e.g. print(my_seq.lstrip("-"))

See also the strip and rstrip methods.

reverse_complement()
Return the reverse complement sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_dna = Seq("CCCCCGATAGNR", IUPAC.ambiguous_dna)
>>> my_dna
Seq('CCCCCGATAGNR', IUPACAmbiguousDNA())
```

(continues on next page)

(continued from previous page)

```
>>> my_dna.reverse_complement()
Seq('YNCTATCGGGGG', IUPACAmbiguousDNA())
```

Note in the above example, since R = G or A, its complement is Y (which denotes C or T).

You can of course used mixed case sequences,

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("CCCCCgatA-G", generic_dna)
>>> my_dna
Seq('CCCCCgatA-G', DNAAlphabet())
>>> my_dna.reverse_complement()
Seq('C-TatcGGGGG', DNAAlphabet())
```

Trying to complement a protein sequence raises an exception:

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.reverse_complement()
Traceback (most recent call last):
   ...
ValueError: Proteins do not have complements!
```

rfind(*sub*, *start=0*, *end=9223372036854775807*)

Find from right method, like that of a python string.

This behaves like the python string method of the same name.

Returns an integer, the index of the last (right most) occurrence of substring argument sub in the (sub)sequence given by [start:end].

Arguments:

- sub - a string or another Seq object to look for
- start - optional integer, slice start
- end - optional integer, slice end

Returns -1 if the subsequence is NOT found.

e.g. Locating the last typical start codon, AUG, in an RNA sequence:

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.rfind("AUG")
15
```

rsplit(*sep=None*, *maxsplit=-1*)

Do a right split method, like that of a python string.

This behaves like the python string method of the same name.

Return a list of the 'words' in the string (as Seq objects), using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done COUNTING FROM THE RIGHT. If maxsplit is omitted, all splits are made.

Following the python string method, sep will by default be any white space (tabs, spaces, newlines) but this is unlikely to apply to biological sequences.

e.g. print(my_seq.rsplit("*",1))

See also the split method.

rstrip (*chars=None*)
Return a new Seq object with trailing (right) end stripped.

This behaves like the python string method of the same name.

Optional argument chars defines which characters to remove. If omitted or None (default) then as for the python string method, this defaults to removing any white space.

e.g. Removing a nucleotide sequence's polyadenylation (poly-A tail):

```
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
>>> my_seq = Seq("CGGTACGCTTATGTCACGTAGAAAAAA", IUPAC.unambiguous_dna)
>>> my_seq
Seq('CGGTACGCTTATGTCACGTAGAAAAAA', IUPACUnambiguousDNA())
>>> my_seq.rstrip("A")
Seq('CGGTACGCTTATGTCACGTAG', IUPACUnambiguousDNA())
```

See also the strip and lstrip methods.

split (*sep=None*, *maxsplit=-1*)
Split method, like that of a python string.

This behaves like the python string method of the same name.

Return a list of the 'words' in the string (as Seq objects), using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If maxsplit is omitted, all splits are made.

Following the python string method, sep will by default be any white space (tabs, spaces, newlines) but this is unlikely to apply to biological sequences.

e.g.

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_aa = my_rna.translate()
>>> my_aa
Seq('VMAIVMGR*KGAR*L', HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> for pep in my_aa.split("*"):
...     pep
Seq('VMAIVMGR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('KGAR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('L', HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> for pep in my_aa.split("*", 1):
...     pep
Seq('VMAIVMGR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('KGAR*L', HasStopCodon(ExtendedIUPACProtein(), '*'))
```

See also the rsplit method:

```
>>> for pep in my_aa.rsplit("*", 1):
...     pep
Seq('VMAIVMGR*KGAR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('L', HasStopCodon(ExtendedIUPACProtein(), '*'))
```

startswith (*prefix*, *start=0*, *end=9223372036854775807*)
Return True if the Seq starts with the given prefix, False otherwise.

This behaves like the python string method of the same name.

Return True if the sequence starts with the specified prefix (a string or another Seq object), False otherwise. With optional start, test sequence beginning at that position. With optional end, stop comparing sequence at that position. prefix can also be a tuple of strings to try. e.g.

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.startswith("GUC")
True
>>> my_rna.startswith("AUG")
False
>>> my_rna.startswith("AUG", 3)
True
>>> my_rna.startswith(("UCC", "UCA", "UCG"), 1)
True
```

strip(*chars=None*)

Return a new Seq object with leading and trailing ends stripped.

This behaves like the python string method of the same name.

Optional argument chars defines which characters to remove. If omitted or None (default) then as for the python string method, this defaults to removing any white space.

e.g. print(my_seq.strip("-"))

See also the lstrip and rstrip methods.

tomutable()

Return the full sequence as a MutableSeq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("MKQHKAMIVALIVICITAVVAAL",
...              IUPAC.protein)
>>> my_seq
Seq('MKQHKAMIVALIVICITAVVAAL', IUPACProtein())
>>> my_seq.tomutable()
MutableSeq('MKQHKAMIVALIVICITAVVAAL', IUPACProtein())
```

Note that the alphabet is preserved.

tostring()

Return the full sequence as a python string (DEPRECATED).

You are now encouraged to use str(my_seq) instead of my_seq.tostring().

transcribe()

Return the RNA sequence from a DNA sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG",
...                  IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> coding_dna.transcribe()
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
```

Trying to transcribe a protein or RNA sequence raises an exception:

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.transcribe()
Traceback (most recent call last):
   ...
ValueError: Proteins cannot be transcribed!
```

translate(*table='Standard'*, *stop_symbol='*'*, *to_stop=False*, *cds=False*, *gap=None*)

Turn a nucleotide sequence into a protein sequence by creating a new Seq object.

This method will translate DNA or RNA sequences, and those with a nucleotide or generic alphabet. Trying to translate a protein sequence raises an exception.

Arguments:

- table - Which codon table to use? This can be either a name (string), an NCBI identifier (integer), or a CodonTable object (useful for non-standard genetic codes). This defaults to the "Standard" table.
- stop_symbol - Single character string, what to use for terminators. This defaults to the asterisk, "*".
- to_stop - Boolean, defaults to False meaning do a full translation continuing on past any stop codons (translated as the specified stop_symbol). If True, translation is terminated at the first in frame stop codon (and the stop_symbol is not appended to the returned protein sequence).
- cds - Boolean, indicates this is a complete CDS. If True, this checks the sequence starts with a valid alternative start codon (which will be translated as methionine, M), that the sequence length is a multiple of three, and that there is a single in frame stop codon at the end (this will be excluded from the protein sequence, regardless of the to_stop option). If these tests fail, an exception is raised.
- gap - Single character string to denote symbol used for gaps. It will try to guess the gap character from the alphabet.

e.g. Using the standard table:

```
>>> coding_dna = Seq("GTGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
>>> coding_dna.translate()
Seq('VAIVMGR*KGAR*', HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> coding_dna.translate(stop_symbol="@")
Seq('VAIVMGR@KGAR@', HasStopCodon(ExtendedIUPACProtein(), '@'))
>>> coding_dna.translate(to_stop=True)
Seq('VAIVMGR', ExtendedIUPACProtein())
```

Now using NCBI table 2, where TGA is not a stop codon:

```
>>> coding_dna.translate(table=2)
Seq('VAIVMGRWKGAR*', HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> coding_dna.translate(table=2, to_stop=True)
Seq('VAIVMGRWKGAR', ExtendedIUPACProtein())
```

In fact, GTG is an alternative start codon under NCBI table 2, meaning this sequence could be a complete CDS:

```
>>> coding_dna.translate(table=2, cds=True)
Seq('MAIVMGRWKGAR', ExtendedIUPACProtein())
```

It isn't a valid CDS under NCBI table 1, due to both the start codon and also the in frame stop codons:

```
>>> coding_dna.translate(table=1, cds=True)
Traceback (most recent call last):
    ...
TranslationError: First codon 'GTG' is not a start codon
```

If the sequence has no in-frame stop codon, then the to_stop argument has no effect:

```
>>> coding_dna2 = Seq("TTGGCCATTGTAATGGGCCGC")
>>> coding_dna2.translate()
Seq('LAIVMGR', ExtendedIUPACProtein())
>>> coding_dna2.translate(to_stop=True)
Seq('LAIVMGR', ExtendedIUPACProtein())
```

When translating gapped sequences, the gap character is inferred from the alphabet:

```
>>> from Bio.Alphabet import Gapped
>>> coding_dna3 = Seq("GTG---GCCATT", Gapped(IUPAC.unambiguous_dna))
>>> coding_dna3.translate()
Seq('V-AI', Gapped(ExtendedIUPACProtein(), '-'))
```

It is possible to pass the gap character when the alphabet is missing:

```
>>> coding_dna4 = Seq("GTG---GCCATT")
>>> coding_dna4.translate(gap='-')
Seq('V-AI', Gapped(ExtendedIUPACProtein(), '-'))
```

NOTE - Ambiguous codons like "TAN" or "NNN" could be an amino acid or a stop codon. These are translated as "X". Any invalid codon (e.g. "TA?" or "T-A") will throw a TranslationError.

NOTE - This does NOT behave like the python string's translate method. For that use str(my_seq).translate(...) instead.

ungap (*gap=None*)

Return a copy of the sequence without the gap character(s).

The gap character can be specified in two ways - either as an explicit argument, or via the sequence's alphabet. For example:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("-ATA--TGAAAT-TTGAAAA", generic_dna)
>>> my_dna
Seq('-ATA--TGAAAT-TTGAAAA', DNAAlphabet())
>>> my_dna.ungap("-")
Seq('ATATGAAATTTGAAAA', DNAAlphabet())
```

If the gap character is not given as an argument, it will be taken from the sequence's alphabet (if defined). Notice that the returned sequence's alphabet is adjusted since it no longer requires a gapped alphabet:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC, Gapped, HasStopCodon
>>> my_pro = Seq("MVVLE=AD*", HasStopCodon(Gapped(IUPAC.protein, "=")))
>>> my_pro
Seq('MVVLE=AD*', HasStopCodon(Gapped(IUPACProtein(), '='), '*'))
>>> my_pro.ungap()
Seq('MVVLEAD*', HasStopCodon(IUPACProtein(), '*'))
```

Or, with a simpler gapped DNA example:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC, Gapped
>>> my_seq = Seq("CGGGTAG=AAAAAA", Gapped(IUPAC.unambiguous_dna, "="))
>>> my_seq
Seq('CGGGTAG=AAAAAA', Gapped(IUPACUnambiguousDNA(), '='))
>>> my_seq.ungap()
Seq('CGGGTAGAAAAAA', IUPACUnambiguousDNA())
```

As long as it is consistent with the alphabet, although it is redundant, you can still supply the gap character as an argument to this method:

```
>>> my_seq
Seq('CGGGTAG=AAAAAA', Gapped(IUPACUnambiguousDNA(), '='))
>>> my_seq.ungap("=")
Seq('CGGGTAGAAAAAA', IUPACUnambiguousDNA())
```

However, if the gap character given as the argument disagrees with that declared in the alphabet, an exception is raised:

```
>>> my_seq
Seq('CGGGTAG=AAAAAA', Gapped(IUPACUnambiguousDNA(), '='))
>>> my_seq.ungap("-")
Traceback (most recent call last):
   ...
ValueError: Gap '-' does not match '=' from alphabet
```

Finally, if a gap character is not supplied, and the alphabet does not define one, an exception is raised:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("ATA--TGAAAT-TTGAAAA", generic_dna)
>>> my_dna
Seq('ATA--TGAAAT-TTGAAAA', DNAAlphabet())
>>> my_dna.ungap()
Traceback (most recent call last):
   ...
ValueError: Gap character not given and not defined in alphabet
```

upper()

Return an upper case copy of the sequence.

```
>>> from Bio.Alphabet import HasStopCodon, generic_protein
>>> from Bio.Seq import Seq
>>> my_seq = Seq("VHLTPeeK*", HasStopCodon(generic_protein))
>>> my_seq
Seq('VHLTPeeK*', HasStopCodon(ProteinAlphabet(), '*'))
>>> my_seq.lower()
Seq('vhltpeek*', HasStopCodon(ProteinAlphabet(), '*'))
>>> my_seq.upper()
Seq('VHLTPEEK*', HasStopCodon(ProteinAlphabet(), '*'))
```

This will adjust the alphabet if required. See also the lower method.

1.1.5 Core.Protein

class Fred2.Core.Protein.**Protein**(*_seq*, *gene_id='unknown'*, *transcript_id=None*, *orig_transcript=None*, *vars=None*)

Bases: *Fred2.Core.Base.MetadataLogger*, Bio.Seq.Seq

Protein corresponding to exactly one transcript.

Note: For accessing and manipulating the sequence see also Bio.Seq.Seq (from Biopython)

---

back_transcribe()

Return the DNA sequence from an RNA sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG",
...                     IUPAC.unambiguous_rna)
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
>>> messenger_rna.back_transcribe()
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
```

Trying to back-transcribe a protein or DNA sequence raises an exception:

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.back_transcribe()
Traceback (most recent call last):
   ...
ValueError: Proteins cannot be back transcribed!
```

complement()

Return the complement sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_dna = Seq("CCCCCGATAG", IUPAC.unambiguous_dna)
>>> my_dna
Seq('CCCCCGATAG', IUPACUnambiguousDNA())
>>> my_dna.complement()
Seq('GGGGGCTATC', IUPACUnambiguousDNA())
```

You can of course used mixed case sequences,

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("CCCCCgatA-GD", generic_dna)
>>> my_dna
Seq('CCCCCgatA-GD', DNAAlphabet())
>>> my_dna.complement()
Seq('GGGGGctaT-CH', DNAAlphabet())
```

Note in the above example, ambiguous character D denotes G, A or T so its complement is H (for C, T or A).

Trying to complement a protein sequence raises an exception.

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.complement()
Traceback (most recent call last):
   ...
ValueError: Proteins do not have complements!
```

count (*sub*, *start=0*, *end=9223372036854775807*)

Return a non-overlapping count, like that of a python string.

This behaves like the python string method of the same name, which does a non-overlapping count!

For an overlapping search use the newer count_overlap() method.

Returns an integer, the number of occurrences of substring argument sub in the (sub)sequence given by [start:end]. Optional arguments start and end are interpreted as in slice notation.

Arguments:

- sub - a string or another Seq object to look for
- start - optional integer, slice start
- end - optional integer, slice end

e.g.

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AAAATGA")
>>> print(my_seq.count("A"))
5
>>> print(my_seq.count("ATG"))
1
>>> print(my_seq.count(Seq("AT")))
1
>>> print(my_seq.count("AT", 2, -1))
1
```

HOWEVER, please note because python strings and Seq objects (and MutableSeq objects) do a non-overlapping search, this may not give the answer you expect:

```
>>> "AAAA".count("AA")
2
>>> print(Seq("AAAA").count("AA"))
2
```

An overlapping search, as implemented in .count_overlap(), would give the answer as three!

count_overlap (*sub*, *start=0*, *end=9223372036854775807*)

Return an overlapping count.

For a non-overlapping search use the count() method.

Returns an integer, the number of occurrences of substring argument sub in the (sub)sequence given by [start:end]. Optional arguments start and end are interpreted as in slice notation.

Arguments:

- sub - a string or another Seq object to look for
- start - optional integer, slice start
- end - optional integer, slice end

e.g.

```
>>> from Bio.Seq import Seq
>>> print(Seq("AAAA").count_overlap("AA"))
3
>>> print(Seq("ATATATATA").count_overlap("ATA"))
4
>>> print(Seq("ATATATATA").count_overlap("ATA", 3, -1))
1
```

Where substrings do not overlap, should behave the same as the count() method:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AAAATGA")
>>> print(my_seq.count_overlap("A"))
5
>>> my_seq.count_overlap("A") == my_seq.count("A")
True
>>> print(my_seq.count_overlap("ATG"))
1
>>> my_seq.count_overlap("ATG") == my_seq.count("ATG")
True
>>> print(my_seq.count_overlap(Seq("AT")))
1
>>> my_seq.count_overlap(Seq("AT")) == my_seq.count(Seq("AT"))
True
>>> print(my_seq.count_overlap("AT", 2, -1))
1
>>> my_seq.count_overlap("AT", 2, -1) == my_seq.count("AT", 2, -1)
True
```

HOWEVER, do not use this method for such cases because the count() method is much for efficient.

endswith (*suffix*, *start=0*, *end=9223372036854775807*)

Return True if the Seq ends with the given suffix, False otherwise.

This behaves like the python string method of the same name.

Return True if the sequence ends with the specified suffix (a string or another Seq object), False otherwise. With optional start, test sequence beginning at that position. With optional end, stop comparing sequence at that position. suffix can also be a tuple of strings to try. e.g.

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.endswith("UUG")
True
>>> my_rna.endswith("AUG")
False
>>> my_rna.endswith("AUG", 0, 18)
True
>>> my_rna.endswith(("UCC", "UCA", "UUG"))
True
```

find (*sub*, *start=0*, *end=9223372036854775807*)

Find method, like that of a python string.

This behaves like the python string method of the same name.

Returns an integer, the index of the first occurrence of substring argument sub in the (sub)sequence given by [start:end].

Arguments:

- sub - a string or another Seq object to look for
- start - optional integer, slice start
- end - optional integer, slice end

Returns -1 if the subsequence is NOT found.

e.g. Locating the first typical start codon, AUG, in an RNA sequence:

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.find("AUG")
3
```

get_metadata (*label*, *only_first=False*)

Getter for the saved metadata with the key `label`

Parameters

- **label** (*str*) – key for the metadata that is inferred
- **only_first** (*bool*) – true if only the the first element of the matadata list is to be returned

log_metadata (*label*, *value*)

Inserts a new metadata

Parameters

- **label** (*str*) – key for the metadata that will be added
- **value** (*list(object)*) – any kindy of additional value that should be kept

lower ()

Return a lower case copy of the sequence.

This will adjust the alphabet if required. Note that the IUPAC alphabets are upper case only, and thus a generic alphabet must be substituted.

```
>>> from Bio.Alphabet import Gapped, generic_dna
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
>>> my_seq = Seq("CGGTACGCTTATGTCACGTAG*AAAAAA",
...              Gapped(IUPAC.unambiguous_dna, "*"))
>>> my_seq
Seq('CGGTACGCTTATGTCACGTAG*AAAAAA', Gapped(IUPACUnambiguousDNA(), '*'))
>>> my_seq.lower()
Seq('cggtacgcttatgtcacgtag*aaaaaa', Gapped(DNAAlphabet(), '*'))
```

See also the upper method.

lstrip (*chars=None*)

Return a new Seq object with leading (left) end stripped.

This behaves like the python string method of the same name.

Optional argument chars defines which characters to remove. If omitted or None (default) then as for the python string method, this defaults to removing any white space.

e.g. print(my_seq.lstrip("-"))

See also the strip and rstrip methods.

newid = <method-wrapper 'next' of itertools.count object>

reverse_complement()
Return the reverse complement sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_dna = Seq("CCCCCGATAGNR", IUPAC.ambiguous_dna)
>>> my_dna
Seq('CCCCCGATAGNR', IUPACAmbiguousDNA())
>>> my_dna.reverse_complement()
Seq('YNCTATCGGGGG', IUPACAmbiguousDNA())
```

Note in the above example, since R = G or A, its complement is Y (which denotes C or T).

You can of course used mixed case sequences,

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("CCCCCgatA-G", generic_dna)
>>> my_dna
Seq('CCCCCgatA-G', DNAAlphabet())
>>> my_dna.reverse_complement()
Seq('C-TatcGGGGG', DNAAlphabet())
```

Trying to complement a protein sequence raises an exception:

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.reverse_complement()
Traceback (most recent call last):
   ...
ValueError: Proteins do not have complements!
```

rfind(*sub*, *start=0*, *end=9223372036854775807*)
Find from right method, like that of a python string.

This behaves like the python string method of the same name.

Returns an integer, the index of the last (right most) occurrence of substring argument sub in the (sub)sequence given by [start:end].

Arguments:

- sub - a string or another Seq object to look for
- start - optional integer, slice start
- end - optional integer, slice end

Returns -1 if the subsequence is NOT found.

e.g. Locating the last typical start codon, AUG, in an RNA sequence:

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.rfind("AUG")
15
```

rsplit(*sep=None*, *maxsplit=-1*)
Do a right split method, like that of a python string.

This behaves like the python string method of the same name.

Return a list of the 'words' in the string (as Seq objects), using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done COUNTING FROM THE RIGHT. If maxsplit is omitted, all splits are made.

Following the python string method, sep will by default be any white space (tabs, spaces, newlines) but this is unlikely to apply to biological sequences.

e.g. print(my_seq.rsplit("*",1))

See also the split method.

rstrip (*chars=None*)

Return a new Seq object with trailing (right) end stripped.

This behaves like the python string method of the same name.

Optional argument chars defines which characters to remove. If omitted or None (default) then as for the python string method, this defaults to removing any white space.

e.g. Removing a nucleotide sequence's polyadenylation (poly-A tail):

```
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
>>> my_seq = Seq("CGGTACGCTTATGTCACGTAGAAAAAA", IUPAC.unambiguous_dna)
>>> my_seq
Seq('CGGTACGCTTATGTCACGTAGAAAAAA', IUPACUnambiguousDNA())
>>> my_seq.rstrip("A")
Seq('CGGTACGCTTATGTCACGTAG', IUPACUnambiguousDNA())
```

See also the strip and lstrip methods.

split (*sep=None*, *maxsplit=-1*)

Split method, like that of a python string.

This behaves like the python string method of the same name.

Return a list of the 'words' in the string (as Seq objects), using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If maxsplit is omitted, all splits are made.

Following the python string method, sep will by default be any white space (tabs, spaces, newlines) but this is unlikely to apply to biological sequences.

e.g.

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_aa = my_rna.translate()
>>> my_aa
Seq('VMAIVMGR*KGAR*L', HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> for pep in my_aa.split("*"):
...     pep
Seq('VMAIVMGR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('KGAR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('L', HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> for pep in my_aa.split("*", 1):
...     pep
Seq('VMAIVMGR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('KGAR*L', HasStopCodon(ExtendedIUPACProtein(), '*'))
```

See also the rsplit method:

```
>>> for pep in my_aa.rsplit("*", 1):
...     pep
Seq('VMAIVMGR*KGAR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('L', HasStopCodon(ExtendedIUPACProtein(), '*'))
```

startswith(*prefix*, *start=0*, *end=9223372036854775807*)
Return True if the Seq starts with the given prefix, False otherwise.

This behaves like the python string method of the same name.

Return True if the sequence starts with the specified prefix (a string or another Seq object), False otherwise. With optional start, test sequence beginning at that position. With optional end, stop comparing sequence at that position. prefix can also be a tuple of strings to try. e.g.

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.startswith("GUC")
True
>>> my_rna.startswith("AUG")
False
>>> my_rna.startswith("AUG", 3)
True
>>> my_rna.startswith(("UCC", "UCA", "UCG"), 1)
True
```

strip(*chars=None*)
Return a new Seq object with leading and trailing ends stripped.

This behaves like the python string method of the same name.

Optional argument chars defines which characters to remove. If omitted or None (default) then as for the python string method, this defaults to removing any white space.

e.g. print(my_seq.strip("-"))

See also the lstrip and rstrip methods.

tomutable()
Return the full sequence as a MutableSeq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("MKQHKAMIVALIVICITAVVAAL",
...              IUPAC.protein)
>>> my_seq
Seq('MKQHKAMIVALIVICITAVVAAL', IUPACProtein())
>>> my_seq.tomutable()
MutableSeq('MKQHKAMIVALIVICITAVVAAL', IUPACProtein())
```

Note that the alphabet is preserved.

tostring()
Return the full sequence as a python string (DEPRECATED).

You are now encouraged to use str(my_seq) instead of my_seq.tostring().

transcribe()
Return the RNA sequence from a DNA sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG",
...                  IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> coding_dna.transcribe()
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
```

Trying to transcribe a protein or RNA sequence raises an exception:

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.transcribe()
Traceback (most recent call last):
   ...
ValueError: Proteins cannot be transcribed!
```

translate(*table='Standard'*, *stop_symbol='*'*, *to_stop=False*, *cds=False*, *gap=None*)

Turn a nucleotide sequence into a protein sequence by creating a new Seq object.

This method will translate DNA or RNA sequences, and those with a nucleotide or generic alphabet. Trying to translate a protein sequence raises an exception.

Arguments:

- table - Which codon table to use? This can be either a name (string), an NCBI identifier (integer), or a CodonTable object (useful for non-standard genetic codes). This defaults to the "Standard" table.
- stop_symbol - Single character string, what to use for terminators. This defaults to the asterisk, "*".
- to_stop - Boolean, defaults to False meaning do a full translation continuing on past any stop codons (translated as the specified stop_symbol). If True, translation is terminated at the first in frame stop codon (and the stop_symbol is not appended to the returned protein sequence).
- cds - Boolean, indicates this is a complete CDS. If True, this checks the sequence starts with a valid alternative start codon (which will be translated as methionine, M), that the sequence length is a multiple of three, and that there is a single in frame stop codon at the end (this will be excluded from the protein sequence, regardless of the to_stop option). If these tests fail, an exception is raised.
- gap - Single character string to denote symbol used for gaps. It will try to guess the gap character from the alphabet.

e.g. Using the standard table:

```
>>> coding_dna = Seq("GTGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
>>> coding_dna.translate()
Seq('VAIVMGR*KGAR*', HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> coding_dna.translate(stop_symbol="@")
Seq('VAIVMGR@KGAR@', HasStopCodon(ExtendedIUPACProtein(), '@'))
>>> coding_dna.translate(to_stop=True)
Seq('VAIVMGR', ExtendedIUPACProtein())
```

Now using NCBI table 2, where TGA is not a stop codon:

```
>>> coding_dna.translate(table=2)
Seq('VAIVMGRWKGAR*', HasStopCodon(ExtendedIUPACProtein(), '*'))
```

(continues on next page)

(continued from previous page)

```
>>> coding_dna.translate(table=2, to_stop=True)
Seq('VAIVMGRWKGAR', ExtendedIUPACProtein())
```

In fact, GTG is an alternative start codon under NCBI table 2, meaning this sequence could be a complete CDS:

```
>>> coding_dna.translate(table=2, cds=True)
Seq('MAIVMGRWKGAR', ExtendedIUPACProtein())
```

It isn't a valid CDS under NCBI table 1, due to both the start codon and also the in frame stop codons:

```
>>> coding_dna.translate(table=1, cds=True)
Traceback (most recent call last):
    ...
TranslationError: First codon 'GTG' is not a start codon
```

If the sequence has no in-frame stop codon, then the to_stop argument has no effect:

```
>>> coding_dna2 = Seq("TTGGCCATTGTAATGGGCCGC")
>>> coding_dna2.translate()
Seq('LAIVMGR', ExtendedIUPACProtein())
>>> coding_dna2.translate(to_stop=True)
Seq('LAIVMGR', ExtendedIUPACProtein())
```

When translating gapped sequences, the gap character is inferred from the alphabet:

```
>>> from Bio.Alphabet import Gapped
>>> coding_dna3 = Seq("GTG---GCCATT", Gapped(IUPAC.unambiguous_dna))
>>> coding_dna3.translate()
Seq('V-AI', Gapped(ExtendedIUPACProtein(), '-'))
```

It is possible to pass the gap character when the alphabet is missing:

```
>>> coding_dna4 = Seq("GTG---GCCATT")
>>> coding_dna4.translate(gap='-')
Seq('V-AI', Gapped(ExtendedIUPACProtein(), '-'))
```

NOTE - Ambiguous codons like "TAN" or "NNN" could be an amino acid or a stop codon. These are translated as "X". Any invalid codon (e.g. "TA?" or "T-A") will throw a TranslationError.

NOTE - This does NOT behave like the python string's translate method. For that use str(my_seq).translate(...) instead.

ungap (*gap=None*)

Return a copy of the sequence without the gap character(s).

The gap character can be specified in two ways - either as an explicit argument, or via the sequence's alphabet. For example:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("-ATA--TGAAAT-TTGAAAA", generic_dna)
>>> my_dna
Seq('-ATA--TGAAAT-TTGAAAA', DNAAlphabet())
>>> my_dna.ungap("-")
Seq('ATATGAAATTTGAAAA', DNAAlphabet())
```

If the gap character is not given as an argument, it will be taken from the sequence's alphabet (if defined). Notice that the returned sequence's alphabet is adjusted since it no longer requires a gapped alphabet:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC, Gapped, HasStopCodon
>>> my_pro = Seq("MVVLE=AD*", HasStopCodon(Gapped(IUPAC.protein, "=")))
>>> my_pro
Seq('MVVLE=AD*', HasStopCodon(Gapped(IUPACProtein(), '='), '*'))
>>> my_pro.ungap()
Seq('MVVLEAD*', HasStopCodon(IUPACProtein(), '*'))
```

Or, with a simpler gapped DNA example:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC, Gapped
>>> my_seq = Seq("CGGGTAG=AAAAAA", Gapped(IUPAC.unambiguous_dna, "="))
>>> my_seq
Seq('CGGGTAG=AAAAAA', Gapped(IUPACUnambiguousDNA(), '='))
>>> my_seq.ungap()
Seq('CGGGTAGAAAAAA', IUPACUnambiguousDNA())
```

As long as it is consistent with the alphabet, although it is redundant, you can still supply the gap character as an argument to this method:

```
>>> my_seq
Seq('CGGGTAG=AAAAAA', Gapped(IUPACUnambiguousDNA(), '='))
>>> my_seq.ungap("=")
Seq('CGGGTAGAAAAAA', IUPACUnambiguousDNA())
```

However, if the gap character given as the argument disagrees with that declared in the alphabet, an exception is raised:

```
>>> my_seq
Seq('CGGGTAG=AAAAAA', Gapped(IUPACUnambiguousDNA(), '='))
>>> my_seq.ungap("-")
Traceback (most recent call last):
   ...
ValueError: Gap '-' does not match '=' from alphabet
```

Finally, if a gap character is not supplied, and the alphabet does not define one, an exception is raised:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("ATA--TGAAAT-TTGAAAA", generic_dna)
>>> my_dna
Seq('ATA--TGAAAT-TTGAAAA', DNAAlphabet())
>>> my_dna.ungap()
Traceback (most recent call last):
   ...
ValueError: Gap character not given and not defined in alphabet
```

upper()

Return an upper case copy of the sequence.

```
>>> from Bio.Alphabet import HasStopCodon, generic_protein
>>> from Bio.Seq import Seq
>>> my_seq = Seq("VHLTPeeK*", HasStopCodon(generic_protein))
```

(continues on next page)

(continued from previous page)

```
>>> my_seq
Seq('VHLTPeeK*', HasStopCodon(ProteinAlphabet(), '*'))
>>> my_seq.lower()
Seq('vhltpeek*', HasStopCodon(ProteinAlphabet(), '*'))
>>> my_seq.upper()
Seq('VHLTPEEK*', HasStopCodon(ProteinAlphabet(), '*'))
```

This will adjust the alphabet if required. See also the lower method.

1.1.6 Core.Result

class Fred2.Core.Result.**AResult**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

Bases: pandas.core.frame.DataFrame

A *AResult* object is a pandas.DataFrame with with multi-indexing.

This class is used as interface and can be extended with custom short-cuts for the sometimes often tedious calls in pandas

T

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
                0     1
name        Alice   Bob
score         9.5     8
employed    False  True
kids            0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name          object
score        float64
employed        bool
kids           int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

abs()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

abs Series/DataFrame containing the absolute value of each element.

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
```

(continues on next page)

(continued from previous page)

```
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
    a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
    a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

numpy.absolute : calculate the absolute value element-wise.

add(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.radd

add_prefix(*prefix*)

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

prefix [str] The string to add before each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_suffix: Suffix row labels with string *suffix*. DataFrame.add_suffix: Suffix column labels with string *suffix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
```

(continues on next page)

(continued from previous page)

```
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0       1       3
1       2       4
2       3       5
3       4       6
```

add_suffix (*suffix*)

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

suffix [str] The string to add after each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_prefix: Prefix row labels with string *prefix*. DataFrame.add_prefix: Prefix column labels with string *prefix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

agg (*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

aggregate (*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

align(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)

Align two objects on their axes with the specified join method for each axis Index

other : DataFrame or Series join : {'outer', 'inner', 'left', 'right'}, default 'outer' axis : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None)

level [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level

copy [boolean, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

method : str, default None limit : int, default None fill_axis : {0 or 'index', 1 or 'columns'}, default 0

Filling axis, method and limit

broadcast_axis [{0 or 'index', 1 or 'columns'}, default None] Broadcast values along this axis, if aligning two objects of different dimensions

(left, right) [(DataFrame, type of other)] Aligned objects

all (*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether all elements are True, potentially over an axis.

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

all : Series or DataFrame (if level specified)

pandas.Series.all : Return True if all elements are True pandas.DataFrame.any : Return True if one (or more) elements are True

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

any(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

any : Series or DataFrame (if level specified)

pandas.DataFrame.all : Return whether all elements are True.

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([True, False]).any()
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
```

(continues on next page)

(continued from previous page)

```
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

append(*other*, *ignore_index=False*, *verify_integrity=False*, *sort=None*)

Append rows of *other* to the end of this frame, returning a new object. Columns not in this frame are added as new columns.

other [DataFrame or Series/dict-like object, or list of these] The data to append.

ignore_index [boolean, default False] If True, do not use the index labels.

verify_integrity [boolean, default False] If True, raise ValueError on creating index with duplicates.

sort [boolean, default None] Sort columns if the columns of *self* and *other* are not aligned. The default sorting is deprecated and will change to not-sorting in a future version of pandas. Explicitly pass `sort=True` to silence the warning and sort. Explicitly pass `sort=False` to silence the warning and not sort.

New in version 0.23.0.

appended : DataFrame

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

pandas.concat [General function to concatenate DataFrame, Series] or Panel objects

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With *ignore_index* set to True:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
...     df = df.append({'A': i}, ignore_index=True)
>>> df
   A
0  0
1  1
2  2
3  3
4  4
```

More efficient:

```
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
...           ignore_index=True)
   A
0  0
1  1
2  2
3  3
4  4
```

apply (*func*, *axis=0*, *broadcast=None*, *raw=False*, *reduce=None*, *result_type=None*, *args=()*, ***kwds*)
Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

func [function] Function to apply to each column or row.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

broadcast [bool, optional] Only relevant for aggregation functions:

- `False` or `None` : returns a Series whose length is the length of the index or the number of columns (based on the *axis* parameter)
- `True` : results will be broadcast to the original shape of the frame, the original index and columns will be retained.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by result_type='broadcast'.

raw [bool, default False]

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

reduce [bool or None, default None] Try to apply reduction procedures. If the DataFrame is empty, *apply* will use *reduce* to determine whether the result should be a Series or a DataFrame. If `reduce=None` (the default), *apply*'s return value will be guessed by calling *func* on an empty Series (note: while guessing, exceptions raised by *func* will be ignored). If `reduce=True` a Series will always be returned, and if `reduce=False` a DataFrame will always be returned.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by `result_type='reduce'`.

result_type [{'expand', 'reduce', 'broadcast', None}, default None] These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.
- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

New in version 0.23.0.

args [tuple] Positional arguments to pass to *func* in addition to the array/series.

****kwds** Additional keyword arguments to pass as keywords arguments to *func*.

In the current implementation apply calls *func* twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if *func* has side-effects, as they will take effect twice for the first column/row.

DataFrame.applymap: For elementwise operations DataFrame.aggregate: only perform aggregating type operations DataFrame.transform: only perform transformating type operations

```
>>> df = pd.DataFrame([[4, 9],] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
```

(continues on next page)

(continued from previous page)

```
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Retuning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing result_type='expand' will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
```

(continues on next page)

(continued from previous page)

```
1  1  2
2  1  2
```

applied : Series or DataFrame

applymap(*func*)

Apply a function to a Dataframe elementwise.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

func [callable] Python function, returns a single value from a single value.

DataFrame Transformed DataFrame.

DataFrame.apply : Apply a function along input axis of DataFrame

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.applymap(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.applymap(lambda x: x**2)
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

But it's better to avoid applymap in that case.

```
>>> df ** 2
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

as_blocks(*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

copy : boolean, default True

values : a dict of dtype -> Constructor Types

as_matrix(*columns=None*)

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

columns: list, optional, default:None If None, return all columns, otherwise, returns specified columns.

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a flot64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

pandas.DataFrame.values

asfreq(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)
Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

freq : DateOffset object, or string method : {'backfill'/'bfill', 'pad'/'ffill'}, default None

Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill

how [{'start', 'end'}, default end] For PeriodIndex only, see PeriodIndex.asfreq

normalize [bool, default False] Whether to reset output index to midnight

fill_value: scalar, optional Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

converted : type of caller

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:03:00    3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                     s
2000-01-01 00:00:00    0.0
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    2.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    3.0
2000-01-01 00:03:00    3.0
```

reindex

To learn more about the frequency strings, please see this link.

asof (*where*, *subset=None*)

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

where : date or array of dates subset : string or list of strings, default None

if not None use these columns for NaN propagation

Dates are assumed to be sorted Raises if this is not the case

where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

where is Index: same shape object as input

merge_asof

assign (***kwargs*)

Assign new columns to a DataFrame, returning a new object (a copy) with the new columns added to the original ones. Existing columns that are re-assigned will be overwritten.

kwargs [keyword, value pairs] keywords are the column names. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

df [DataFrame] A new DataFrame with the new columns in addition to all the existing columns.

Assigning multiple columns within the same `assign` is possible. For Python 3.6 and above, later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order. For Python 3.5 and below, the order of keyword arguments is not specified, you cannot refer to newly created or modified columns. All items are computed first, and then assigned in alphabetical order.

Changed in version 0.23.0: Keyword argument order is maintained for Python 3.6 and later.

```
>>> df = pd.DataFrame({'A': range(1, 11), 'B': np.random.randn(10)})
```

Where the value is a callable, evaluated on *df*:

```
>>> df.assign(ln_A = lambda x: np.log(x.A))
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the value already exists and is inserted:

```
>>> newcol = np.log(df['A'])
>>> df.assign(ln_A=newcol)
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the keyword arguments depend on each other

```
>>> df = pd.DataFrame({'A': [1, 2, 3]})
```

```
>>> df.assign(B=df.A, C=lambda x:x['A']+ x['B'])
   A  B  C
 0  1  1  2
 1  2  2  4
 2  3  3  6
```

astype(***kwargs*)

Cast a pandas object to a specified dtype `dtype`.

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy [bool, default True.] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors [{'raise', 'ignore'}, default 'raise'.] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

raise_on_error [raise on invalid input] Deprecated since version 0.20.0: Use `errors` instead

kwargs : keyword arguments to pass on to the constructor

casted : type of caller

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> ser.astype('category', ordered=True, categories=[2, 1])
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to a numeric type. numpy.ndarray.astype : Cast a numpy array to a specified type.

at

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.iat [Access a single value for a row/column pair by integer] position

DataFrame.loc : Access a group of rows and columns by label(s) Series.at : Access a single value using a label

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

KeyError When label does not exist in DataFrame

at_time (*time*, *asof=False*)

Select values at particular time of day (e.g. 9:30AM).

TypeError If the index is not a `DatetimeIndex`

time : datetime.time or string

values_at_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

between_time : Select values between particular times of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_at_time : Get just the index locations for

values at particular time of the day

axes

Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['col1', 'col2'],
dtype='object')]
```

between_time (*start_time*, *end_time*, *include_start=True*, *include_end=True*)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

TypeError If the index is not a `DatetimeIndex`

start_time : datetime.time or string end_time : datetime.time or string include_start : boolean, default True include_end : boolean, default True

values_between_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

at_time : Select values at a particular time of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_between_time : Get just the index locations for

values between particular times of the day

bfill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='bfill')`

blocks
Internal property, property synonym for as_blocks()

Deprecated since version 0.21.0.

bool()
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

boxplot(*column=None*, *by=None*, *ax=None*, *fontsize=None*, *rot=0*, *grid=True*, *figsize=None*, *layout=None*, *return_type=None*, ***kwds*)
Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to *1.5 * IQR (IQR = Q3 - Q1)* from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia's entry for boxplot.

column [str or list of str, optional] Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.

by [str or array-like, optional] Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in *by*.

ax [object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.

fontsize [float or str] Tick label font size in points or as a string (e.g., *large*).

rot [int or float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate sytem.

grid [boolean, default True] Setting this to True will show the grid.

figsize [A tuple (width, height) in inches] The size of the figure to create in matplotlib.

layout [tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 columns and 5 rows, starting from the top-left.

return_type [{'axes', 'dict', 'both'} or None, default 'axes'] The kind of object to return. The default is `axes`.

- 'axes' returns the matplotlib axes the boxplot is drawn on.
- 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot.
- 'both' returns a namedtuple with the axes and dict.
- when grouping with `by`, a Series mapping columns to `return_type` is returned.

  If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

result :

The return type depends on the *return_type* parameter:

- 'axes' : object of class matplotlib.axes.Axes

- 'dict' : dict of matplotlib.lines.Line2D objects
- 'both' : a nametuple with strucure (ax, lines)

For data grouped with `by`:

- `Series`
- `array` (for `return_type = None`)

Series.plot.hist: Make a histogram. matplotlib.pyplot.boxplot : Matplotlib equivalent plot.

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`. For instance:

A list of strings (i.e. `['X', 'Y']`) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

The layout of boxplot can be adjusted giving a tuple to `layout`:

Additional formatting can be done to the boxplot, like suppressing the grid (`grid=False`), rotating the labels in the x-axis (i.e. `rot=45`) or changing the fontsize (i.e. `fontsize=15`):

The parameter `return_type` can be used to select the type of element returned by *boxplot*. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```
>>> boxplot = df.boxplot(column=['Col1','Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

When grouping with `by`, a Series mapping columns to `return_type` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned:

```
>>> boxplot =  df.boxplot(column=['Col1', 'Col2'], by='X',
...                       return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

clip (*lower=None*, *upper=None*, *axis=None*, *inplace=False*, **args*, ***kwargs*)
Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

lower [float or array_like, default None] Minimum threshold value. All values below this threshold will be set to it.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

clip_lower : Clip values below specified threshold(s). clip_upper : Clip values above specified threshold(s).

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

clip_lower(*threshold*, *axis=None*, *inplace=False*)

Return copy of the input with values below a threshold truncated.

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Series.clip [Return copy of input with values below and above] thresholds truncated.

Series.clip_upper [Return copy of input with values above] threshold truncated.

clipped : same type as input

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

clip_upper (*threshold*, *axis=None*, *inplace=False*)

Return copy of input with values above given value(s) truncated.

threshold : float or array_like axis : int or string axis name, optional

Align object with threshold along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data

New in version 0.21.0.

clip

clipped : same type as input

columns

The column labels of the DataFrame.

combine (*other*, *func*, *fill_value=None*, *overwrite=True*)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

other : DataFrame func : function

Function that takes two series as inputs and return a Series or a scalar

fill_value : scalar value overwrite : boolean, default True

If True then overwrite values for common keys in the calling frame

result : DataFrame

```
>>> df1 = DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, lambda s1, s2: s1 if s1.sum() < s2.sum() else s2)
   A  B
0  0  3
1  0  3
```

DataFrame.combine_first [Combine two DataFrame objects and default to] non-null values in frame calling the method

combine_first (*other*)

Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

other : DataFrame

combined : DataFrame

df1's values prioritized, use values from df2 to fill holes:

```
>>> df1 = pd.DataFrame([[1, np.nan]])
>>> df2 = pd.DataFrame([[3, 4]])
>>> df1.combine_first(df2)
   0    1
0  1  4.0
```

DataFrame.combine [Perform series-wise operation on two DataFrames] using a given function

compound(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

compounded : Series or DataFrame (if level specified)

consolidate(*inplace=False*)

Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

convert_objects(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Return a fixed frequency timedelta index,

with day as the default.

converted : same as input object

copy(*deep=True*)

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

deep [bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

copy [Series, DataFrame or Panel] Object type matches caller.

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
```

(continues on next page)

(continued from previous page)

```
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

corr (*method='pearson'*, *min_periods=1*)

Compute pairwise correlation of columns, excluding NA/null values

method [{'pearson', 'kendall', 'spearman'}]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation

y : DataFrame

corrwith (*other*, *axis=0*, *drop=False*)

Compute pairwise correlation between rows or columns of two DataFrame objects.

other : DataFrame, Series axis : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' to compute column-wise, 1 or 'columns' for row-wise

drop [boolean, default False] Drop missing indices from result, default returns union of all

correls : Series

count (*axis=0*, *level=None*, *numeric_only=False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, and optionally *numpy.inf* (depending on *pandas.options.mode.use_inf_as_na*) are considered NA.

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each **row**.

level [int or str, optional] If the axis is a *MultiIndex* (hierarchical), count along a particular *level*, collapsing into a *DataFrame*. A *str* specifies the level name.

numeric_only [boolean, default False] Include only *float*, *int* or *boolean* data.

Series or DataFrame For each column/row the number of non-NA/null entries. If *level* is specified returns a *DataFrame*.

Series.count: number of non-NA elements in a Series DataFrame.shape: number of DataFrame rows and columns (including NA

elements)

DataFrame.isna: boolean same-sized DataFrame showing places of NA elements

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                    ["John", "Myla", None, "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> df
   Person   Age  Single
0    John  24.0   False
1    Myla   NaN    True
2    None  21.0    True
3    John  33.0    True
4    Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    4
Age       4
Single    5
dtype: int64
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    2
3    3
4    3
dtype: int64
```

Counts for one level of a *MultiIndex*:

```
>>> df.set_index(["Person", "Single"]).count(level="Person")
        Age
Person
John      2
Myla      1
```

cov (*min_periods=None*)

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as `NaN`.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result.

DataFrame The covariance matrix of the series of the DataFrame.

pandas.Series.cov : compute covariance with another Series pandas.core.window.EWM.cov: expoential weighted sample covariance pandas.core.window.Expanding.cov : expanding sample covariance pandas.core.window.Rolling.cov : rolling sample covariance

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                   columns=['dogs', 'cats'])
>>> df.cov()
          dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
          a         b         c         d         e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
```

(continues on next page)

(continued from previous page)

```
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
          a         b         c
a  0.316741       NaN -0.150812
b       NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

cummax (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummax : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.max [Similar functionality] but ignores `NaN` values.

DataFrame.max [Return the maximum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cummin (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummin : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
```

(continues on next page)

(continued from previous page)

```
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.min [Similar functionality] but ignores `NaN` values.

DataFrame.min [Return the minimum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumprod (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumprod : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.prod [Similar functionality] but ignores `NaN` values.

DataFrame.prod [Return the product over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumsum(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumsum : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.sum [Similar functionality] but ignores `NaN` values.

DataFrame.sum [Return the sum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

`describe` (*percentiles=None*, *include=None*, *exclude=None*)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

summary: Series/DataFrame of summary statistics

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                       3
unique                      2
top       2010-01-01 00:00:00
freq                        2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                     'numeric': [1, 2, 3],
...                     'categorical': pd.Categorical(['d','e','f'])
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
        categorical  numeric object
count             3      3.0      3
unique            3      NaN      3
top               f      NaN      c
freq              1      NaN      1
mean            NaN      2.0    NaN
std             NaN      1.0    NaN
min             NaN      1.0    NaN
25%             NaN      1.5    NaN
50%             NaN      2.0    NaN
75%             NaN      2.5    NaN
max             NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
        categorical  numeric
count            3      3.0
```

(continues on next page)

(continued from previous page)

```
unique            3      NaN
top               f      NaN
freq              1      NaN
mean            NaN      2.0
std             NaN      1.0
min             NaN      1.0
25%             NaN      1.5
50%             NaN      2.0
75%             NaN      2.5
max             NaN      3.0
```

DataFrame.count DataFrame.max DataFrame.min DataFrame.mean DataFrame.std DataFrame.select_dtypes

diff (*periods=1*, *axis=0*)

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

periods [int, default 1] Periods to shift for calculating difference, accepts negative values.

axis [{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

New in version 0.16.1..

diffed : DataFrame

Series.diff: First discrete difference for a Series. DataFrame.pct_change: Percent change over given number of periods. DataFrame.shift: Shift index by desired number of periods with an

optional time freq.

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a    b     c
0 NaN  0.0   0.0
1 NaN -1.0   3.0
2 NaN -1.0   7.0
3 NaN -1.0  13.0
4 NaN  0.0  20.0
5 NaN  2.0  28.0
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

div(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

divide(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

dot (*other*)

Matrix multiplication with DataFrame or Series objects. Can also be called using *self @ other* in Python >= 3.5.

other : DataFrame or Series

dot_product : DataFrame or Series

drop (*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

labels [single label or list-like] Index or column labels to drop.

axis [{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index, columns [single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

New in version 0.21.0.

level [int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace [bool, default False] If True, do operation inplace and return None.

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

dropped : pandas.DataFrame

DataFrame.loc : Label-location based indexer for selection by label. DataFrame.dropna : Return DataFrame with labels on given axis omitted

where (all or any) data are missing

DataFrame.drop_duplicates [Return DataFrame with duplicate rows] removed, optionally only considering certain columns

Series.drop : Return Series with specified index labels removed.

KeyError If none of the labels are found in the selected axis

```
>>> df = pd.DataFrame(np.arange(12).reshape(3,4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                              [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3,0.2]])
>>> df
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
lama    speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
```

(continues on next page)

(continued from previous page)

```
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

drop_duplicates (*subset=None*, *keep='first'*, *inplace=False*)

Return DataFrame with duplicate rows removed, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Drop duplicates except for the first occurrence.
- `last` : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

inplace [boolean, default False] Whether to drop duplicates in place or to return a copy

deduplicated : DataFrame

dropna (*axis=0*, *how='any'*, *thresh=None*, *subset=None*, *inplace=False*)

Remove missing values.

See the User Guide for more on which values are considered missing, and how to work with missing data.

axis [{0 or 'index', 1 or 'columns'}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Deprecated since version 0.23.0:: Pass tuple or list to drop on multiple

axes.

how [{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

thresh [int, optional] Require that many non-NA values.

subset [array-like, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace [bool, default False] If True, do operation inplace and return None.

DataFrame DataFrame with NA entries dropped from it.

DataFrame.isna: Indicate missing values. DataFrame.notna : Indicate existing (non-missing) values. DataFrame.fillna : Replace missing values. Series.dropna : Drop missing values. Index.dropna : Drop missing indices.

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'born'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type of each column.

pandas.DataFrame.ftypes : dtype and sparsity information.

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
string              object
dtype: object
```

duplicated(*subset=None*, *keep='first'*)

Return boolean Series denoting duplicate rows, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Mark duplicates as `True` except for the first occurrence.
- `last` : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

duplicated : Series

empty

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

bool If DataFrame is empty, return True, if not return False.

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
```

(continues on next page)

(continued from previous page)

```
>>> df.dropna().empty
True
```

pandas.Series.dropna pandas.DataFrame.dropna

eq(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods eq

equals(*other*)
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

eval(*expr*, *inplace=False*, ***kwargs*)
Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

expr [str] The expression string to evaluate.

inplace [bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

New in version 0.18.0..

kwargs [dict] See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

ndarray, scalar, or pandas object The result of the evaluation.

DataFrame.query [Evaluates a boolean expression to query the columns] of a frame.

DataFrame.assign [Can evaluate an expression or function to create new] values for a column.

pandas.eval [Evaluate a Python expression as a string using various] backends.

For more details see the API documentation for `eval()`. For detailed examples see enhancing performance with eval.

```
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
>>> df.eval('A + B')
0    11
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
```

Use `inplace=True` to modify the original DataFrame.

```
>>> df.eval('C = A + B', inplace=True)
>>> df
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
```

ewm(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*)
Provides exponential weighted functions

New in version 0.18.0.

com [float, optional] Specify decay in terms of center of mass, $\alpha = 1/(1 + com),$ for $com \geq 0$

span [float, optional] Specify decay in terms of span, $\alpha = 2/(span + 1),$ for $span \geq 1$

halflife [float, optional] Specify decay in terms of half-life, $\alpha = 1 - exp(log(0.5)/halflife),$ for $halflife > 0$

alpha [float, optional] Specify smoothing factor α directly, $0 < \alpha \leq 1$

New in version 0.18.0.

min_periods [int, default 0] Minimum number of observations in window required to have a value (otherwise result is NA).

adjust [boolean, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

ignore_na [boolean, default False] Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), . . . , 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

rolling : Provides rolling window calculations expanding : Provides expanding transformations.

expanding (*min_periods=1*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

min_periods [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

center [boolean, default False] Set the labels at the center of the window.

axis : int or string, default 0

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

rolling : Provides rolling window calculations ewm : Provides exponential weighted functions

ffill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='ffill')`

fillna(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)
Fill NA/NaN values using the specified method

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis : {0 or 'index', 1 or 'columns'} inplace : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

interpolate : Fill NaN values using interpolation. reindex, asfreq

filled : DataFrame

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 2.0 1
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

filter(*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

items [list-like] List of info axis to restrict to (must not all be present)

like [string] Keep info axis where "arg in col == True"

regex [string (regular expression)] Keep info axis with re.search(regex, col) == True

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame

same type as input object

```
>>> df
one  two  three
mouse     1    2      3
rabbit    4    5      6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
one  two  three
rabbit    4    5      6
```

pandas.DataFrame.loc

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

filter_result(*expressions*)

Filter result based on a list of expressions

Parameters **expressions** (*list((str,comparator,float))*) – A list of triples consisting of (method_name, comparator, threshold)

Returns A new filtered AResult object

Return type *AResult*

first(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

subset : type of caller

last : Select final periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

first_valid_index()

Return index for first non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

floordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rfloordiv

classmethod from_csv(*path*, *header=0*, *sep=','*, *','*, *index_col=0*, *parse_dates=True*, *encoding=None*, *tupleize_cols=None*, *infer_datetime_format=False*)

Read CSV file.

Deprecated since version 0.21.0: Use `pandas.read_csv()` instead.

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a DataFrame of time series data.

This method only differs from the preferred `pandas.read_csv()` in some defaults:

- *index_col* is `0` instead of `None` (take first column as index by default)
- *parse_dates* is `True` instead of `False` (try parsing the index as datetime by default)

So a `pd.DataFrame.from_csv(path)` can be replaced by `pd.read_csv(path, index_col=0, parse_dates=True)`.

path : string file path or file handle / StringIO header : int, default 0

Row to use as header (skip prior rows)

sep [string, default ','] Field delimiter

index_col [int or sequence, default 0] Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

parse_dates [boolean, default True] Parse dates. Different default from read_table

tupleize_cols [boolean, default False] write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

infer_datetime_format: boolean, default False If True and *parse_dates* is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

pandas.read_csv

y : DataFrame

classmethod from_dict(*data*, *orient='columns'*, *dtype=None*, *columns=None*)
Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

data [dict] Of the form {field : array-like} or {field : dict}.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'.

dtype [dtype, default None] Data type to force, otherwise infer.

columns [list, default None] Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'`.

New in version 0.23.0.

pandas.DataFrame

DataFrame.from_records [DataFrame from ndarray (structured] dtype), list of tuples, dict, or DataFrame

DataFrame : DataFrame object creation using constructor

By default the keys of the dict become the DataFrame columns:

```
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Specify `orient='index'` to create the DataFrame using dictionary keys as rows:

```
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
       0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                        columns=['A', 'B', 'C', 'D'])
       A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
```

classmethod from_items(*items*, *columns=None*, *orient='columns'*)
Construct a dataframe from a list of tuples

Deprecated since version 0.23.0: *from_items* is deprecated and will be removed in a future version. Use `DataFrame.from_dict(dict(items))` instead. `DataFrame.from_dict(OrderedDict(items))` may be used to preserve the key order.

Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the columns, but depends on the specified orientation). The values should be arrays or Series.

items [sequence of (key, value) pairs] Values should be arrays or Series.

columns [sequence of column labels, optional] Must be passed if orient='index'.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the input correspond to column labels, pass 'columns' (default). Otherwise if the keys correspond to the index, pass 'index'.

frame : DataFrame

classmethod from_records(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float=False*, *nrows=None*)

Convert structured or record ndarray to DataFrame

data : ndarray (structured dtype), list of tuples, dict, or DataFrame index : string, list of fields, array-like

Field of array to use as the index, alternately a specific set of input labels to use

exclude [sequence, default None] Columns or fields to exclude

columns [sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

coerce_float [boolean, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

df : DataFrame

ftypes

Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type and indication of sparse/dense of each column.

pandas.DataFrame.dtypes: Series with just dtype information. pandas.SparseDataFrame : Container for sparse tabular data.

Sparse data should have the same dtypes as its dense representation.

```
>>> import numpy as np
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).ftypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).ftypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

ge(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods ge

get(*key*, *default=None*)
Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

key : object

value : type of items contained in object

get_dtype_counts()
Return counts of unique dtypes in this object.

dtype [Series] Series with the count of columns with each dtype.

dtypes : Return the dtypes in this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

get_ftype_counts()
Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

ftypes [Return ftypes (indication of sparse/dense and dtype) in] this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

get_value(*index*, *col*, *takeable=False*)
Quickly retrieve single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label takeable : interpret the index/col as indexers, default False

value : scalar value

get_values()

Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a *pandas.SparseArray*, the data are first converted to a dense representation.

numpy.ndarray Numpy representation of DataFrame

values : Numpy representation of DataFrame. pandas.SparseArray : Container for sparse data.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a      b    c
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
     a    c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

groupby(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=False*, *observed=False*, ***kwargs*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

by [mapping, function, label, or list of labels] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted a (single) key.

axis : int, default 0 level : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

as_index [boolean, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort [boolean, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.

group_keys [boolean, default True] When calling apply, add group keys to index to identify pieces

squeeze [boolean, default False] reduce the dimensionality of the return type if possible, otherwise return a consistent type

observed [boolean, default False] This only applies if any of the groupers are Categoricals If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

GroupBy object

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

See the user guide for more.

resample [Convenience method for frequency conversion and resampling] of time series.

gt (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods gt

head (*n=5*)
Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

n [int, default 5] Number of rows to select.

obj_head [type of caller] The first *n* rows of the caller object.

pandas.DataFrame.tail: Returns the last *n* rows.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
```

(continues on next page)

(continued from previous page)

```
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

hist(*column=None*, *by=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *ax=None*, *sharex=False*, *sharey=False*, *figsize=None*, *layout=None*, *bins=10*, ***kwds*)
Make a histogram of the DataFrame's.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

data [DataFrame] The pandas object holding the data.

column [string or sequence] If passed, will be used to limit data to a subset of columns.

by [object, optional] If passed, then used to form histograms for separate groups.

grid [boolean, default True] Whether to show axis grid lines.

xlabelsize [int, default None] If specified changes the x-axis label size.

xrot [float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize [int, default None] If specified changes the y-axis label size.

yrot [float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

ax [Matplotlib axes object, default None] The axes to plot the histogram on.

sharex [boolean, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey [boolean, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize [tuple] The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

layout [tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

axes : matplotlib.AxesSubplot or numpy.ndarray of them

matplotlib.pyplot.hist : Plot a histogram using matplotlib.

iat

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.at : Access a single value for a row/column label pair DataFrame.loc : Access a group of rows and columns by label(s) DataFrame.iloc : Access a group of rows and columns by integer position(s)

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

IndexError When integer position is out of bounds

idxmax (*axis=0*, *skipna=True*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmax : Series

This method is the DataFrame version of `ndarray.argmax`.

Series.idxmax

idxmin (*axis=0*, *skipna=True*)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmin : Series

This method is the DataFrame version of `ndarray.argmin`.

Series.idxmin

iloc

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at Selection by Position

index

The index (row labels) of the DataFrame.

infer_objects()

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to numeric typeR

converted : same type as input object

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

info(*verbose=None*, *buf=None*, *max_cols=None*, *memory_usage=None*, *null_counts=None*)
Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

verbose [bool, optional] Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

buf [writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

max_cols [int, optional] When to switch from the verbose to the truncated output. If the DataFrame has more than *max_cols* columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

memory_usage [bool, str, optional] Specifies whether total memory usage of the DataFrame elements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources.

null_counts [bool, optional] Whether to show the non-null counts. By default, this is shown only if the frame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

None This method prints a summary of a DataFrame and returns None.

DataFrame.describe: Generate descriptive statistics of DataFrame columns.

DataFrame.memory_usage: Memory usage of DataFrame columns.

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                    "float_col": float_values})
>>> df
   int_col text_col  float_col
0        1    alpha       0.00
1        2     beta       0.25
2        3    gamma       0.50
3        4    delta       0.75
4        5  epsilon       1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
int_col      5 non-null int64
```

(continues on next page)

(continued from previous page)

```
text_col     5 non-null object
float_col    5 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w", encoding="utf-8") as f:
...     f.write(s)
260
```

The *memory_usage* parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 22.9+ MB
```

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 188.8 MB
```

insert (*loc*, *column*, *value*, *allow_duplicates=False*)

Insert column into DataFrame at specified location.

Raises a ValueError if *column* is already contained in the DataFrame, unless *allow_duplicates* is set to

True.

loc [int] Insertion index. Must verify 0 <= loc <= len(columns)

column [string, number, or hashable object] label of the inserted column

value : int, Series, or array-like allow_duplicates : bool, optional

interpolate(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

method [{'linear', 'time', 'index', 'values', 'nearest', 'zero',]

'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to `scipy.interpolate.interp1d`. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
- 'from_derivatives' refers to BPoly.from_derivatives which replaces 'piecewise_polynomial' interpolation method in scipy 0.18

New in version 0.18.1: Added support for the 'akima' method Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in scipy 0.18; backwards-compatible with scipy < 0.18

axis [{0, 1}, default 0]

- 0: fill column-by-column
- 1: fill row-by-row

limit [int, default None.] Maximum number of consecutive NaNs to fill. Must be greater than 0.

limit_direction : {'forward', 'backward', 'both'}, default 'forward' limit_area : {'inside', 'outside'}, default None

- None: (default) no fill restriction
- 'inside' Only fill NaNs surrounded by valid values (interpolate).
- 'outside' Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

inplace [bool, default False] Update the NDFrame in place if possible.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

kwargs : keyword arguments to pass on to the interpolating function.

Series or DataFrame of same shape interpolated at the NaNs

reindex, replace, fillna

Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

is_copy

isin(*values*)

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

values [iterable, Series, DataFrame or dictionary] The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dictionary, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

DataFrame of booleans

When `values` is a list:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
       A      B
0   True   True
1  False  False
2   True  False
```

When `values` is a dict:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
       A      B
0   True  False  # Note that B didn't match the 1 here.
1  False   True
2   True   True
```

When `values` is a Series or DataFrame:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
       A      B
0   True  False
1  False  False  # Column A in `other` has a 3, but not at index 1.
2   True   True
```

isna()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

isnull()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

items()

Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iteritems()

Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iterrows()

Iterate over DataFrame rows as (index, Series) pairs.

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
```

(continues on next page)

(continued from previous page)

```
>>> print(df['int'].dtype)
int64
```

To preserve dtypes while iterating over the rows, it is better to use *itertuples()* which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

it [generator] A generator that iterates over the rows of the frame.

itertuples : Iterate over DataFrame rows as namedtuples of the values. iteritems : Iterate over (column name, Series) pairs.

itertuples (*index=True*, *name='Pandas'*)

Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.

index [boolean, default True] If True, return the index as the first element of the tuple.

name [string, default "Pandas"] The name of the returned namedtuples or None to return regular tuples.

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. iteritems : Iterate over (column name, Series) pairs.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [0.1, 0.2]},
                      index=['a', 'b'])
>>> df
   col1  col2
a     1   0.1
b     2   0.2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='a', col1=1, col2=0.10000000000000001)
Pandas(Index='b', col1=2, col2=0.20000000000000001)
```

ix

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at Advanced Indexing.

join (*other*, *on=None*, *how='left'*, *lsuffix=''*, *rsuffix=''*, *sort=False*)

Join columns with other DataFrame either on index or on a key column. Efficiently Join multiple DataFrame objects by index at once by passing a list.

other [DataFrame, Series with name field set, or list of DataFrame] Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame

on [name, tuple/list of names, or array-like] Column or index level name(s) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

how [{'left', 'right', 'outer', 'inner'}, default: 'left'] How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use other frame's index
- outer: form union of calling frame's index (or column if on is specified) with other frame's index, and sort it lexicographically
- inner: form intersection of calling frame's index (or column if on is specified) with other frame's index, preserving the order of the calling's one

lsuffix [string] Suffix to use from left frame's overlapping columns

rsuffix [string] Suffix to use from right frame's overlapping columns

sort [boolean, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword)

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

Support for specifying index levels as the *on* parameter was added in version 0.23.0

```
>>> caller = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                        'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> caller
    A key
0  A0  K0
1  A1  K1
2  A2  K2
3  A3  K3
4  A4  K4
5  A5  K5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
    B key
0  B0  K0
1  B1  K1
2  B2  K2
```

Join DataFrames using their indexes.

```
>>> caller.join(other, lsuffix='_caller', rsuffix='_other')
```

```
>>>     A key_caller    B key_other
    0  A0         K0   B0        K0
    1  A1         K1   B1        K1
```

(continues on next page)

(continued from previous page)

```
    2  A2          K2   B2         K2
    3  A3          K3  NaN        NaN
    4  A4          K4  NaN        NaN
    5  A5          K5  NaN        NaN
```

If we want to join using the key columns, we need to set key to be the index in both caller and other. The joined DataFrame will have key as its index.

```
>>> caller.set_index('key').join(other.set_index('key'))
```

```
>>>      A    B
    key
    K0   A0   B0
    K1   A1   B1
    K2   A2   B2
    K3   A3  NaN
    K4   A4  NaN
    K5   A5  NaN
```

Another option to join using the key columns is to use the on parameter. DataFrame.join always uses other's index but we can use any column in the caller. This method preserves the original caller's index in the result.

```
>>> caller.join(other.set_index('key'), on='key')
```

```
>>>     A key    B
    0  A0  K0   B0
    1  A1  K1   B1
    2  A2  K2   B2
    3  A3  K3  NaN
    4  A4  K4  NaN
    5  A5  K5  NaN
```

DataFrame.merge : For column(s)-on-columns(s) operations

joined : DataFrame

keys()

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

kurt(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

kurtosis (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

last (*offset*)
Convenience method for subsetting final periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

subset : type of caller

first : Select initial periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

last_valid_index ()
Return index for last non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

le (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods le

loc
Access a group of rows and columns by label(s) or a boolean array.

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

> **Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at Selection by Label

DataFrame.at : Access a single value for a row/column label pair DataFrame.iloc : Access group of rows and columns by integer position(s) DataFrame.xs : Returns a cross-section (row(s) or column(s)) from the

Series/DataFrame.

Series.loc : Access group of values using labels

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=['cobra', 'viper', 'sidewinder'],
...       columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
```

(continues on next page)

(continued from previous page)

```
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

KeyError: when any items are not found

lookup(*row_labels*, *col_labels*)

Label-based "fancy indexing" function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

row_labels [sequence] The row labels to use for lookup

col_labels [sequence] The column labels to use for lookup

Akin to:

```
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

values [ndarray] The found values

lt(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods lt

mad(*axis=None*, *skipna=None*, *level=None*)
Return the mean absolute deviation of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mad : Series or DataFrame (if level specified)

mask(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

`DataFrame.where()`

max(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the maximum of the values in the object. If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

max : Series or DataFrame (if level specified)

mean(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mean : Series or DataFrame (if level specified)

median(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

median : Series or DataFrame (if level specified)

melt(*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *col_level=None*)

"Unpivots" a DataFrame from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

New in version 0.20.0.

frame : DataFrame id_vars : tuple, list, or ndarray, optional

Column(s) to use as identifier variables.

value_vars [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name [scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name [scalar, default 'value'] Name to use for the 'value' column.

col_level [int or string, optional] If columns are a MultiIndex then use this level to melt.

melt pivot_table DataFrame.pivot

```
>>> import pandas as pd
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> df.melt(id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

memory_usage(*index=True*, *deep=False*)

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

index [bool, default True] Specifies whether to include the memory usage of the DataFrame's index in returned Series. If `index=True` the memory usage of the index the first item in the output.

deep [bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

sizes [Series] A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

numpy.ndarray.nbytes [Total bytes consumed by the elements of an] ndarray.

Series.memory_usage : Bytes consumed by a Series. pandas.Categorical : Memory-efficient array for string values with

many repeated values.

DataFrame.info : Concise summary of a DataFrame.

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64  complex128 object  bool
0      1      1.0      (1+0j)      1  True
1      1      1.0      (1+0j)      1  True
2      1      1.0      (1+0j)      1  True
3      1      1.0      (1+0j)      1  True
4      1      1.0      (1+0j)      1  True
```

```
>>> df.memory_usage()
Index            80
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64         40000
float64       40000
complex128    80000
object        40000
```

(continues on next page)

(continued from previous page)

```
bool           5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index             80
int64          40000
float64        40000
complex128     80000
object        160000
bool            5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5168
```

merge(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *sort=False*, *suffixes=('_x', '_y')*, *copy=True*, *indicator=False*, *validate=None*)

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

right : DataFrame how : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys

on [label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [boolean, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

right_index [boolean, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index

sort [boolean, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword)

suffixes [2-length sequence (tuple, list, ...)] Suffix to apply to overlapping column names in the left and right side, respectively

copy [boolean, default True] If False, do not copy data unnecessarily

indicator [boolean or string, default False] If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in 'left' DataFrame, "right_only" for observations whose merge key only appears in 'right' DataFrame, and "both" if the observation's merge key is found in both.

validate [string, default None] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

New in version 0.21.0.

Support for specifying index levels as the *on*, *left_on*, and *right_on* parameters was added in version 0.23.0

```
>>> A              >>> B
    lkey value         rkey value
0   foo  1         0   foo  5
1   bar  2         1   bar  6
2   baz  3         2   qux  7
3   foo  4         3   bar  8
```

```
>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0  foo   1        foo   5
1  foo   4        foo   5
2  bar   2        bar   6
3  bar   2        bar   8
4  baz   3        NaN   NaN
5  NaN   NaN      qux   7
```

merged [DataFrame] The output type will the be same as 'left', if it is a subclass of DataFrame.

merge_ordered merge_asof DataFrame.join

merge_results(*others*)

Merges results of the same type and returns a merged result

Parameters **others** (list(*AResult*)/*AResult*) – A (list of) *AResult* object(s) of the same class

Returns A new merged *AResult* object

Return type *AResult*

min(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the minimum of the values in the object. If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min : Series or DataFrame (if level specified)

mod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmod

mode(*axis=0*, *numeric_only=False*)
Gets the mode(s) of each element along the axis selected. Adds a row for each mode per label, fills in gaps with nan.

Note that there could be multiple values returned for the selected axis (when more than one item share the maximum frequency), which is the reason why a dataframe is returned. If you want to impute missing values with the mode in a dataframe `df`, you can just do this: `df.fillna(df.mode().iloc[0])`

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index' : get mode of each column
- 1 or 'columns' : get mode of each row

numeric_only [boolean, default False] if True, only apply to numeric columns

modes : DataFrame (sorted)

```
>>> df = pd.DataFrame({'A': [1, 2, 1, 2, 1, 2, 3]})
>>> df.mode()
   A
0  1
1  2
```

mul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

multiply(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

ndim

Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

ndarray.ndim

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

ne(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods ne

nlargest(*n*, *columns*, *keep='first'*)

Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant.

n [int] Number of rows to return.

columns [label or list of labels] Column label(s) to order by.

keep [{'first', 'last'}, default 'first'] Where there are duplicate values:

- *first* : prioritize the first occurrence(s)
- *last* : prioritize the last occurrence(s)

DataFrame The first *n* rows ordered by the given columns in descending order.

DataFrame.nsmallest [Return the first *n* rows ordered by *columns* in] ascending order.

DataFrame.sort_values : Sort DataFrame by the values DataFrame.head : Return the first *n* rows without re-ordering.

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 10, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df
    a  b    c
0   1  a  1.0
1  10  b  2.0
2   8  d  NaN
3  10  c  3.0
4  -1  e  4.0
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column "a".

```
>>> df.nlargest(3, 'a')
    a  b    c
1  10  b  2.0
3  10  c  3.0
2   8  d  NaN
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'a', keep='last')
    a  b    c
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

To order by the largest values in column "a" and then "c", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['a', 'c'])
    a  b    c
```

(continues on next page)

(continued from previous page)

```
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

Attempting to use `nlargest` on non-numeric dtypes will raise a `TypeError`:

```
>>> df.nlargest(3, 'b')
Traceback (most recent call last):
TypeError: Column 'b' has dtype object, cannot use method 'nlargest'
```

notna()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

notnull()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

nsmallest(*n*, *columns*, *keep='first'*)

Get the rows of a DataFrame sorted by the *n* smallest values of *columns*.

n [int] Number of items to retrieve

columns [list or str] Column name or names to order by

keep [{'first', 'last'}, default 'first'] Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

DataFrame

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 11, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nsmallest(3, 'a')
   a  b   c
4 -1  e   4
0  1  a   1
2  8  d NaN
```

nunique (*axis=0*, *dropna=True*)

Return Series with number of distinct observations over requested axis.

New in version 0.20.0.

axis : {0 or 'index', 1 or 'columns'}, default 0 dropna : boolean, default True

Don't include NaN in the counts.

nunique : Series

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 1, 1]})
>>> df.nunique()
A    3
B    1
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
```

pct_change (*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs** Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

chg [Series or DataFrame] The same type as the calling object.

Series.diff : Compute the difference of two elements in a Series. DataFrame.diff : Compute the difference of two elements in a DataFrame. Series.shift : Shift the index by some number of periods. DataFrame.shift : Shift the index by some number of periods.

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
```

(continues on next page)

(continued from previous page)

```
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                   FR        GR        IT
1980-01-01        NaN       NaN       NaN
1980-02-01   0.013810  0.013684  0.006549
1980-03-01   0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...      '2016': [1769950, 30586265],
...      '2015': [1500923, 40912316],
...      '2014': [1371819, 41403351]},
...      index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pipe (*func*, **args*, ***kwargs*)

Apply func(self, *args, **kwargs)

func [function] function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the NDFrame.

args [iterable, optional] positional arguments passed into `func`.

kwargs [mapping, optional] a dictionary of keyword arguments passed into `func`.

object : the return type of `func`.

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pandas.DataFrame.apply pandas.DataFrame.applymap pandas.Series.map

pivot (*index=None*, *columns=None*, *values=None*)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the User Guide for more on reshaping.

index [string or object, optional] Column to use to make new frame's index. If None, uses existing index.

columns [string or object] Column to use to make new frame's columns.

values [string, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Changed in version 0.23.0: Also accept list of column names.

DataFrame Returns reshaped DataFrame.

ValueError: When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

DataFrame.pivot_table [generalization of pivot that can handle] duplicate values for one index/column pair.

DataFrame.unstack [pivot based on the index values instead of a] column.

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
```

(continues on next page)

(continued from previous page)

```
...                     "baz": [1, 2, 3, 4]})
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

pivot_table(*values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*, *dropna=True*, *margins_name='All'*)

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame

values : column to aggregate, optional index : column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

columns [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

aggfunc [function, list of functions, dict, default numpy.mean] If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions

fill_value [scalar, default None] Value to replace missing values with

margins [boolean, default False] Add all row / columns (e.g. for subtotal / grand totals)

dropna [boolean, default True] Do not include columns whose entries are all NaN

margins_name [string, default 'All'] Name of the row / column that will contain the totals when margins is True.

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
>>> df
     A    B      C  D
0  foo  one  small  1
1  foo  one  large  2
2  foo  one  large  2
```

(continues on next page)

(continued from previous page)

```
3  foo  two  small  3
4  foo  two  small  3
5  bar  one  large  4
6  bar  one  small  5
7  bar  two  small  6
8  bar  two  large  7
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': [min, max, np.mean]})
>>> table
                  D   E
               mean max median min
A   C
bar large  5.500000  16   14.5  13
    small  5.500000  15   14.5  14
foo large  2.000000  10    9.5   9
    small  2.333333  12   11.0   8
```

table : DataFrame

DataFrame.pivot [pivot without aggregation that can handle] non-numeric data

plot

alias of `pandas.plotting._core.FramePlotMethods`

pop (*item*)

Return item and drop from frame. Raise KeyError if not found.

item [str] Column label to be popped

popped : Series

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
```

(continues on next page)

(continued from previous page)

```
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

pow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rpow

prod(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

product (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

quantile(*q=0.5*, *axis=0*, *numeric_only=True*, *interpolation='linear'*)

Return values at the given quantile over requested axis, a la numpy.percentile.

q [float or array-like, default 0.5 (50% quantile)] 0 <= q <= 1, the quantile(s) to compute

axis [{0, 1, 'index', 'columns'} (default 0)] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

numeric_only [boolean, default True] If False, the quantile of datetime and timedelta data will be computed as well

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.18.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

quantiles : Series or DataFrame

- If `q` is an array, a DataFrame will be returned where the index is `q`, the columns are the columns of self, and the values are the quantiles.
- If `q` is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                      columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
                       'B': [pd.Timestamp('2010'),
                             pd.Timestamp('2011')],
                       'C': [pd.Timedelta('1 days'),
                             pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
```

(continues on next page)

(continued from previous page)

```
C        1 days 12:00:00
Name: 0.5, dtype: object
```

pandas.core.window.Rolling.quantile

query(*expr*, *inplace=False*, ***kwargs*)

Query the columns of a frame with a boolean expression.

expr [string] The query string to evaluate. You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

inplace [bool] Whether the query should modify the data in place or return a modified copy

New in version 0.18.0.

kwargs [dict] See the documentation for `pandas.eval()` for complete details on the keyword arguments accepted by `DataFrame.query()`.

q : DataFrame

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level `pandas.eval()` function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the `query` documentation in indexing.

pandas.eval DataFrame.eval

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = pd.DataFrame(randn(10, 2), columns=list('ab'))
>>> df.query('a > b')
>>> df[df.a > df.b]  # same result as the previous expression
```

radd(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.add

rank (*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)
Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values

axis [{0 or 'index', 1 or 'columns'}, default 0] index to direct ranking

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

ranks : same type as caller

rdiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

reindex(***kwargs*)
Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like, optional] New labels / index to conform the axis specified by 'axis' to.

index, columns [array-like, optional (should be specified using keywords)] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis [int or str, optional] Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...       'http_status': [200,200,404,404,301],
...       'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...        index=index)
>>> df
            http_status  response_time
Firefox             200           0.04
Chrome              200           0.02
Safari              404           0.07
IE10                404           0.08
Konqueror           301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
```

(continues on next page)

(continued from previous page)

```
IE10                 404          0.08
Chrome               200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to backpropagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the `method` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29     100
2009-12-30     100
2009-12-31     100
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

Please note that the `NaN` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the `NaN` values present in the original dataframe, use the `fillna()` method.

See the user guide for more.

reindexed : DataFrame

`reindex_axis`(*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=nan*)

Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis : {0 or 'index', 1 or 'columns'} method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional

Method to use for filling holes in reindexed DataFrame:

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

reindex, reindex_like

reindexed : DataFrame

reindex_like(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)
Return an object with matching indices to myself.

other : Object method : string or None copy : boolean, default True limit : int, default None

Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

reindexed : same as input

rename(***kwargs*)
Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the user guide for more.

mapper, index, columns [dict-like or function, optional] dict-like or functions transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

axis [int or str, optional] Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

copy [boolean, default True] Also copy underlying data

inplace [boolean, default False] Whether to return a new DataFrame. If True then value of copy is ignored.

level [int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

renamed : DataFrame

pandas.DataFrame.rename_axis

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
```

(continues on next page)

(continued from previous page)

```
1  2  5
2  3  6
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

rename_axis(*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter the name of the index or columns.

mapper [scalar, list-like, optional] Value to set as the axis name attribute.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis.

copy [boolean, default True] Also copy underlying data.

inplace [boolean, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

renamed [Series, DataFrame, or None] The same type as the caller or None if *inplace* is True.

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

pandas.Series.rename : Alter Series index labels or name pandas.DataFrame.rename : Alter DataFrame index labels or name pandas.Index.rename : Set new names on index

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s.rename_axis("foo")
foo
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")
     A  B
foo
0    1  4
1    2  5
2    3  6
```

```
>>> df.rename_axis("bar", axis="columns")
bar  A  B
0    1  4
1    2  5
2    3  6
```

reorder_levels (*order*, *axis=0*)

Rearrange index levels using input order. May not drop or duplicate levels

order [list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).

axis [int] Where to reorder levels.

type of caller (new object)

replace (*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)

Replace values given in *to_replace* with *value*.

Values of the DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

to_replace [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be `None` to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:

- This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

value [scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace [boolean, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

regex [bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method [{'pad', 'ffill', 'bfill', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

DataFrame.fillna : Fill NA values DataFrame.where : Replace values based on boolean condition Series.str.replace : Simple string replacement.

DataFrame Object after replacement.

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Scalar 'to_replace' and 'value'

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like 'to_replace'

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
2    3
3    3
4    4
dtype: int64
```

dict-like 'to_replace'

```
>>> df.replace({0: 10, 1: 100})
     A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
     A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
     A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression 'to_replace'

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
      A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$':'new', 'foo':'xyz'})
      A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
      A    B
0   new  abc
1   new  new
2  bait  xyz
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
```

(continues on next page)

(continued from previous page)

```
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the pecularities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value=None` and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

resample(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

rule [string] the offset string or object representing target conversion

axis : int, optional, default 0 closed : {'right', 'left'}

Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label [{'right', 'left'}] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention [{'start', 'end', 's', 'e'}] For PeriodIndex only, controls whether to use the start or end of *rule*

kind: {'timestamp', 'period'}, optional Pass 'timestamp' to convert the resulting index to a `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`. By default the input representation is retained.

loffset [timedelta] Adjust the resampled time labels

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

on [string, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [string or int, optional] For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

Resampler object

See the user guide for more.

To learn more about the offset strings, please see this link.

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5] #select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like)+5
```

```
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))
>>> s
2012    1
```

(continues on next page)

(continued from previous page)

```
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using 'start' *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
                     a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                       columns=['a', 'b', 'c', 'd'],
                       index=pd.MultiIndex.from_product([time, [1, 2]])
                       )
>>> df2.resample('3T', level=0).sum()
                     a  b   c   d
2000-01-01 00:00:00  0  6  12  18
2000-01-01 00:03:00  0  4   8  12
```

groupby : Group by mapping, function, label, or list of labels.

reset_index(*level=None*, *drop=False*, *inplace=False*, *col_level=0*, *col_fill=''*)

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level_0' (if 'index' is already taken) will be used.

level [int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default

drop [boolean, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

col_level [int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill [object, default ''] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

resetted : DataFrame

```
>>> df = pd.DataFrame([('bird',    389.0),
...                    ('bird',     24.0),
...                    ('mammal',   80.5),
...                    ('mammal', np.nan)],
...                   index=['falcon', 'parrot', 'lion', 'monkey'],
...                   columns=('class', 'max_speed'))
>>> df
         class  max_speed
falcon    bird      389.0
parrot    bird       24.0
lion    mammal       80.5
monkey  mammal        NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
    class  max_speed
0    bird      389.0
1    bird       24.0
2  mammal       80.5
3  mammal        NaN
```

You can also use *reset_index* with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                    ('bird', 'parrot'),
...                                    ('mammal', 'lion'),
...                                    ('mammal', 'monkey')],
```

(continues on next page)

(continued from previous page)

```
...                                       names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                      ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
...                    ( 24.0, 'fly'),
...                    ( 80.5, 'run'),
...                    (np.nan, 'jump')],
...                   index=index,
...                   columns=columns)
>>> df
               speed species
                 max    type
class  name
bird   falcon  389.0     fly
       parrot   24.0     fly
mammal lion     80.5     run
       monkey    NaN    jump
```

If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
         class  speed species
                  max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
                speed species
         class    max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

When the index is inserted under another level, we can specify under which one with the parameter *col_fill*:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
              species  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

If we specify a nonexistent level for *col_fill*, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
                genus  speed species
                class    max    type
```

(continues on next page)

(continued from previous page)

```
name
falcon          bird  389.0     fly
parrot          bird   24.0     fly
lion          mammal   80.5     run
monkey        mammal    NaN    jump
```

rfloordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.floordiv

rmod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mod

rmul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mul

rolling(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculations.

New in version 0.18.0.

window [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. This is new in 0.19.0

min_periods [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

center [boolean, default False] Set the labels at the center of the window.

win_type [string, default None] Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

on [string, optional] For a DataFrame, column on which to calculate the rolling window, rather than the index

closed [string, default None] Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

axis : int or string, default 0

a Window or Rolling sub-classed for the particular operation

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
     B
0  NaN
1  1.0
```

(continues on next page)

(continued from previous page)

```
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                   index = [pd.Timestamp('20130101 09:00:00'),
...                            pd.Timestamp('20130101 09:00:02'),
...                            pd.Timestamp('20130101 09:00:03'),
...                            pd.Timestamp('20130101 09:00:05'),
...                            pd.Timestamp('20130101 09:00:06')])
```

```
>>> df
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```
>>> df.rolling('2s').sum()
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

To learn more about the offsets & frequency strings, please see this link.

The recognized win_types are:

- boxcar
- triang
- blackman
- hamming
- bartlett
- parzen
- bohman
- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general_gaussian (needs power, width)
- slepian (needs width).

If win_type=None all points are evenly weighted. To learn more about different window types see scipy.signal window functions.

expanding : Provides expanding transformations. ewm : Provides exponential weighted functions

round(*decimals=0*, **args*, ***kwargs*)

Round a DataFrame to a variable number of decimal places.

decimals [int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

```
>>> df = pd.DataFrame(np.random.random([3, 3]),
...     columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
>>> df
                A         B         C
first    0.028208  0.992815  0.173891
second   0.038683  0.645646  0.577595
third    0.877076  0.149370  0.491027
>>> df.round(2)
           A     B     C
first   0.03  0.99  0.17
second  0.04  0.65  0.58
third   0.88  0.15  0.49
>>> df.round({'A': 1, 'C': 2})
          A         B     C
first   0.0  0.992815  0.17
second  0.0  0.645646  0.58
third   0.9  0.149370  0.49
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
          A  B     C
first   0.0  1  0.17
```

(continues on next page)

(continued from previous page)

```
second  0.0  1  0.58
third   0.9  0  0.49
```

DataFrame object

numpy.around Series.round

rpow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.pow

rsub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
```

(continues on next page)

(continued from previous page)

```
...                   index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.sub

rtruediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*)

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

n [int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac [float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace [boolean, optional] Sample with or without replacement. Default = False.

weights [str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. inf and -inf values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

A new object of same type as caller.

Generate an example `Series` and `DataFrame`:

```
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0   -0.038497
1    1.820773
2   -0.972766
3   -1.598270
4   -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
          A         B         C         D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661
```

Next extract a random sample from both of these objects...

3 random elements from the `Series`:

```
>>> s.sample(n=3)
27   -0.994689
55   -1.049016
67   -0.224565
dtype: float64
```

And a random 10% of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.1, replace=True)
           A         B         C         D
35  1.981780  0.142106  1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640  0.217116
40  0.823173 -0.078816  1.009536  1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
6  -0.148339  0.832938  1.787600 -1.383767
```

You can use *random state* for reproducibility:

```
>>> df.sample(random_state=1)
A         B         C         D
37 -2.027662  0.103611  0.237496 -0.165867
43 -0.259323 -0.583426  1.516140 -0.479118
12 -1.686325 -0.579510  0.985195 -0.460286
8   1.167946  0.429082  1.215742 -1.636041
9   1.197475 -0.864188  1.554031 -1.505264
```

select (*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

crit [function] To be called on each index (label). Should return True or False

axis : int

selection : type of caller

select_dtypes(*include=None*, *exclude=None*)

Return a subset of the DataFrame's columns based on the column dtypes.

include, exclude [scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

ValueError

- If both of `include` and `exclude` are empty
- If `include` and `exclude` have overlapping elements
- If any kind of string dtype is passed in.

subset [DataFrame] The subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.

- To select all *numeric* types, use `np.number` or `'number'`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the numpy dtype hierarchy
- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`
- To select timedeltas, use `np.timedelta64`, `'timedelta'` or `'timedelta64'`
- To select Pandas categorical dtypes, use `'category'`
- To select Pandas datetimetz dtypes, use `'datetimetz'` (new in 0.20.0) or `'datetime64[ns, tz]'`

```
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
        a      b  c
0       1   True  1.0
1       2  False  2.0
2       1   True  1.0
3       2  False  2.0
4       1   True  1.0
5       2  False  2.0
```

```
>>> df.select_dtypes(include='bool')
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

```
>>> df.select_dtypes(include=['float64'])
   c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int'])
       b    c
0   True  1.0
1  False  2.0
2   True  1.0
3  False  2.0
4   True  1.0
5  False  2.0
```

sem(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

sem : Series or DataFrame (if level specified)

set_axis(*labels*, *axis=0*, *inplace=None*)
Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

labels [list-like, Index] The values for the new index.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

pandas.DataFrame.rename_axis : Alter the name of the index or columns.

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

set_index (*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

keys : column label or list of column labels / arrays drop : boolean, default True

Delete columns to be used as the new index

append [boolean, default False] Whether to append columns to existing index

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

verify_integrity [boolean, default False] Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale':[55, 40, 84, 31]})
   month  sale  year
0  1      55    2012
1  4      40    2014
2  7      84    2013
3  10     31    2014
```

Set the index to become the 'month' column:

```
>>> df.set_index('month')
       sale  year
month
1      55    2012
4      40    2014
7      84    2013
10     31    2014
```

Create a multi-index using columns 'year' and 'month':

```
>>> df.set_index(['year', 'month'])
            sale
year  month
2012  1     55
2014  4     40
2013  7     84
2014  10    31
```

Create a multi-index using a set of values and a column:

```
>>> df.set_index([[1, 2, 3, 4], 'year'])
         month  sale
   year
1  2012  1      55
2  2014  4      40
3  2013  7      84
4  2014  10     31
```

dataframe : DataFrame

set_value(*index*, *col*, *value*, *takeable=False*)

Put single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label value : scalar value takeable : interpret the index/col as indexers, default False

frame [DataFrame] If label pair is contained, will be reference to calling DataFrame, otherwise a new object

shape

Return a tuple representing the dimensionality of the DataFrame.

ndarray.shape

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

shift (*periods=1*, *freq=None*, *axis=0*)

Shift index by desired number of periods with an optional time freq

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, optional] Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

axis : {0 or 'index', 1 or 'columns'}

If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data.

shifted : DataFrame

size

Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

ndarray.size

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

skew (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased skew over requested axis Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

skew : Series or DataFrame (if level specified)

slice_shift (*periods=1*, *axis=0*)
Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

periods [int] Number of periods to move, can be positive or negative

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

shifted : same type as caller

sort_index (*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)
Sort object by labels (along an axis)

axis : index, columns to direct sorting level : int or level name or list of ints or list of level names

if not None, sort on values in specified index level(s)

ascending [boolean, default True] Sort ascending vs. descending

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining [bool, default True] if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

sorted_obj : DataFrame

sort_values (*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)
Sort by the values along either axis

by [str or list of str] Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels

Changed in version 0.23.0: Allow specifying index or column level names.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis to be sorted

ascending [bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end

sorted_obj : DataFrame

```
>>> df = pd.DataFrame({
...     'col1' : ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2' : [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
```

(continues on next page)

(continued from previous page)

```
... })
>>> df
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
3   NaN  8    4
4   D    7    2
5   C    4    3
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
    col1 col2 col3
1   A    1    1
0   A    2    0
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
    col1 col2 col3
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
3   NaN  8    4
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
    col1 col2 col3
3   NaN  8    4
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
```

sortlevel (*level=0*, *axis=0*, *ascending=True*, *inplace=False*, *sort_remaining=True*)

Sort multilevel index by chosen axis and primary level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order).

Deprecated since version 0.20.0: Use `DataFrame.sort_index()`

level : int axis : {0 or 'index', 1 or 'columns'}, default 0 ascending : boolean, default True inplace : boolean, default False

Sort the DataFrame without creating a new instance

sort_remaining [boolean, default True] Sort by the other levels too.

sorted : DataFrame

DataFrame.sort_index(level=...)

squeeze (*axis=None*)

Squeeze length 1 dimensions.

axis [None, integer or string axis name, optional] The axis to squeeze if 1-sized.

New in version 0.20.0.

scalar if 1-sized, else original object

stack (*level=-1*, *dropna=True*)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

The new index levels are sorted.

level [int, str, list, default -1] Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

dropna [bool, default True] Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of index and column values that are missing from the original dataframe. See Examples section.

DataFrame or Series Stacked dataframe or series.

DataFrame.unstack [Unstack prescribed level(s) from index axis] onto column axis.

DataFrame.pivot [Reshape dataframe from long format to wide] format.

DataFrame.pivot_table [Create a spreadsheet-style pivot table] as a DataFrame.

The function is named by analogy with a collection of books being re-organised from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of of each other (in the index of the dataframe).

Single level columns

```
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
     weight height
cat       0      1
dog       2      3
>>> df_single_level_cols.stack()
cat  weight    0
     height    1
dog  weight    2
     height    3
dtype: int64
```

Multi level columns: simple case

```
>>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol1)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
     weight
         kg    pounds
cat       1         2
dog       2         4
>>> df_multi_level_cols1.stack()
            weight
cat kg           1
    pounds       2
dog kg           2
    pounds       4
```

Missing values

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
    weight height
        kg      m
cat    1.0    2.0
dog    3.0    4.0
>>> df_multi_level_cols2.stack()
        height  weight
cat kg     NaN     1.0
    m      2.0     NaN
dog kg     NaN     3.0
    m      4.0     NaN
```

Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0)
             kg    m
cat height  NaN  2.0
    weight  1.0  NaN
dog height  NaN  4.0
    weight  3.0  NaN
>>> df_multi_level_cols2.stack([0, 1])
cat  height  m     2.0
     weight  kg    1.0
dog  height  m     4.0
     weight  kg    3.0
dtype: float64
```

Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
    weight height
        kg      m
cat    NaN    1.0
dog    2.0    3.0
>>> df_multi_level_cols3.stack(dropna=False)
        height  weight
cat kg     NaN     NaN
    m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
>>> df_multi_level_cols3.stack(dropna=True)
        height  weight
cat m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
```

std(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

std : Series or DataFrame (if level specified)

style
Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

pandas.io.formats.style.Styler

sub (*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

subtract (*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the sum of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

sum : Series or DataFrame (if level specified)

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

swapaxes (*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately

y : same as input

swaplevel (*i=-2*, *j=-1*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

swapped : type of caller (new object)

Changed in version 0.18.1: The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

tail (*n=5*)

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

n [int, default 5] Number of rows to select.

type of caller The last *n* rows of the caller object.

pandas.DataFrame.head : The first *n* rows of the caller object.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

take(*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

indices [array-like] An array of ints indicating which positions to take.

axis [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

taken [type of caller] An array-like containing the elements taken from the object.

DataFrame.loc : Select a subset of a DataFrame by labels. DataFrame.iloc : Select a subset of a DataFrame by positions. numpy.take : Take elements from an array along an axis.

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                    columns=['name', 'class', 'max_speed'],
...                    index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

to_clipboard(*excel=True*, *sep=None*, ***kwargs*)

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

excel [bool, default True]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

sep [str, default `'\t'`] Field delimiter.

****kwargs** These parameters will be passed to DataFrame.to_csv.

DataFrame.to_csv [Write a DataFrame to a comma-separated values] (csv) file.

read_clipboard : Read text from clipboard and pass to read_table.

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

to_csv(*path_or_buf=None*, *sep=', '*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *encoding=None*, *compression=None*, *quoting=None*, *quotechar='"'*, *line_terminator='\n'*, *chunksize=None*, *tupleize_cols=None*, *date_format=None*, *doublequote=True*, *escapechar=None*, *decimal='.'*)

Write DataFrame to a comma-separated values (csv) file

path_or_buf [string or file handle, default None] File path or object, if None is provided the result is returned as a string.

sep [character, default ‘,’] Field delimiter for the output file.

na_rep [string, default ‘’] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R

mode [str] Python write mode, default ‘w’

encoding [string, optional] A string representing the encoding to use in the output file, defaults to ‘ascii’ on Python 2 and ‘utf-8’ on Python 3.

compression [string, optional] A string representing the compression to use in the output file. Allowed values are ‘gzip’, ‘bz2’, ‘zip’, ‘xz’. This input is only used when the first argument is a filename.

line_terminator [string, default `'\n'`] The newline character or character sequence to use in the output file

quoting [optional constant from csv module] defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric

quotechar [string (length 1), default ‘"’] character used to quote fields

doublequote [boolean, default True] Control quoting of *quotechar* inside a field

escapechar [string (length 1), default None] character used to escape *sep* and *quotechar* when appropriate

chunksize [int or None] rows to write at a time

tupleize_cols [boolean, default False] Deprecated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

Write MultiIndex columns as a list of tuples (if True) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if False).

date_format [string, default None] Format string for datetime objects

decimal: string, default '.' Character recognized as decimal separator. E.g. use ',' for European data

to_dense()
Return dense representation of NDFrame (as opposed to sparse)

to_dict(*orient='dict'*, *into=<type 'dict'>*)
Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

orient [str {'dict', 'list', 'series', 'split', 'records', 'index'}] Determines the type of the values of the dictionary.

- 'dict' (default) : dict like {column -> {index -> value}}
- 'list' : dict like {column -> [values]}
- 'series' : dict like {column -> Series(values)}
- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'records' : list like [{column -> value}, ... , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

into [class, default dict] The collections.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

New in version 0.21.0.

result : collections.Mapping like {column -> {index -> value}}

DataFrame.from_dict: create a DataFrame from a dictionary DataFrame.to_json: convert a DataFrame to JSON format

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   col1  col2
a     1   0.50
b     2   0.75
>>> df.to_dict()
{'col1': {'a': 1, 'b': 2}, 'col2': {'a': 0.5, 'b': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': a    1
         b    2
         Name: col1, dtype: int64,
 'col2': a    0.50
         b    0.75
         Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['a', 'b'], 'columns': ['col1', 'col2'],
 'data': [[1.0, 0.5], [2.0, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1.0, 'col2': 0.5}, {'col1': 2.0, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'a': {'col1': 1.0, 'col2': 0.5}, 'b': {'col1': 2.0, 'col2': 0.75}}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('a', 1), ('b', 2)])),
             ('col2', OrderedDict([('a', 0.5), ('b', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1.0, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2.0, 'col2': 0.75})]
```

to_excel(*excel_writer*, *sheet_name='Sheet1'*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *startrow=0*, *startcol=0*, *engine=None*, *merge_cells=True*, *encoding=None*, *inf_rep='inf'*, *verbose=True*, *freeze_panes=None*)

Write DataFrame to an excel sheet

excel_writer [string or ExcelWriter object] File path or existing ExcelWriter

sheet_name [string, default 'Sheet1'] Name of sheet which will contain DataFrame

na_rep [string, default ''] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow : upper left cell row to dump data frame

startcol : upper left cell column to dump data frame

engine [string, default None] write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

merge_cells [boolean, default True] Write MultiIndex and Hierarchical Rows as merged cells.

encoding: string, default None encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep [string, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel)

freeze_panes [tuple of integer (length 2), default None] Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

For compatibility with to_csv, to_excel serializes lists and dicts to strings before writing.

to_feather(*fname*)
write out the binary feather-format for DataFrames

New in version 0.20.0.

fname [str] string file path

to_gbq(*destination_table*, *project_id*, *chunksize=None*, *verbose=None*, *reauth=False*, *if_exists='fail'*, *private_key=None*, *auth_local_webserver=False*, *table_schema=None*)
Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

Authentication to the Google BigQuery service is via OAuth 2.0.

- If `private_key` is provided, the library loads the JSON service account credentials and uses those to authenticate.
- If no `private_key` is provided, the library tries application default credentials.
- If application default credentials are not found or cannot be used with BigQuery, the library authenticates with user account credentials. In this case, you will be asked to grant permissions for product name 'pandas GBQ'.

destination_table [str] Name of table to be written, in the form 'dataset.tablename'.

project_id [str] Google BigQuery Account project ID.

chunksize [int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

reauth [bool, default False] Force Google BigQuery to reauthenticate the user. This is useful if multiple accounts are used.

if_exists [str, default 'fail'] Behavior when the destination table exists. Value can be one of:

'fail' If table exists, do nothing.

'replace' If table exists, drop it, recreate it, and insert data.

'append' If table exists, insert data. Create if does not exist.

private_key [str, optional] Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

auth_local_webserver [bool, default False] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

table_schema [list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'},...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

New in version 0.3.1 of pandas-gbq.

verbose [boolean, deprecated] *Deprecated in Pandas-GBQ 0.4.0.* Use the logging module to adjust verbosity instead.

pandas_gbq.to_gbq : This function in the pandas-gbq library. pandas.read_gbq : Read a DataFrame from Google BigQuery.

to_hdf (*path_or_buf*, *key*, ***kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the user guide.

path_or_buf [str or pandas.HDFStore] File path or HDFStore object.

key [str] Identifier for the group in the store.

mode [{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

format [{'fixed', 'table'}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

append [bool, default False] For Table formats, append the input data to the existing.

data_columns [list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See io.hdf5-query-data-columns. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

errors [str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

DataFrame.read_hdf : Read from HDF file. DataFrame.to_parquet : Write a DataFrame to the binary parquet format. DataFrame.to_sql : Write to a sql table. DataFrame.to_feather : Write out feather-format for DataFrames. DataFrame.to_csv : Write out to a csv file.

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

to_html (*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *bold_rows=True*, *classes=None*, *escape=True*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*, *notebook=False*, *decimal='.'*, *border=None*, *table_id=None*)

Render a DataFrame as an HTML table.

to_html-specific options:

bold_rows [boolean, default True] Make the row labels bold in the output

classes [str or list or tuple, default None] CSS class(es) to apply to the resulting html table

escape [boolean, default True] Convert the characters <, >, and & to HTML-safe sequences.

max_rows [int, optional] Maximum number of rows to show before truncating. If None, show all.

max_cols [int, optional] Maximum number of columns to show before truncating. If None, show all.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe

New in version 0.18.0.

border [int] A `border=border` attribute is included in the opening <*table*> tag. Default `pd.options.html.border`.

New in version 0.19.0.

table_id [str, optional] A css id is included in the opening <*table*> tag if specified.

New in version 0.23.0.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] whether to print column labels, default True

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_json(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression=None*, *index=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

path_or_buf [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'

  - allowed values are: {‘split’,’records’,’index’}
- DataFrame
  - default is ‘columns’
  - allowed values are: {‘split’,’records’,’index’,’columns’,’values’}
- The format of the JSON string
  - ‘split’ : dict like {‘index’ -> [index], ‘columns’ -> [columns], ‘data’ -> [values]}
  - ‘records’ : list like [{column -> value}, … , {column -> value}]
  - ‘index’ : dict like {index -> {column -> value}}
  - ‘columns’ : dict like {column -> {index -> value}}
  - ‘values’ : just the values array
  - ‘table’ : dict like {‘schema’: {schema}, ‘data’: {data}} describing the data, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

date_format [{None, ‘epoch’, ‘iso’}] Type of date conversion. ‘epoch’ = epoch milliseconds, ‘iso’ = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is ‘iso’. For all other orients, the default is ‘epoch’.

double_precision [int, default 10] The number of decimal places to use when encoding floating point values.

force_ascii [boolean, default True] Force encoded string to be ASCII.

date_unit [string, default ‘ms’ (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’, ‘ns’ for second, millisecond, microsecond, and nanosecond respectively.

default_handler [callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines [boolean, default False] If ‘orient’ is ‘records’ write out line delimited json format. Will throw ValueError if incorrect ‘orient’ since others are not list like.

New in version 0.19.0.

compression [{None, ‘gzip’, ‘bz2’, ‘zip’, ‘xz’}] A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

index [boolean, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is ‘split’ or ‘table’.

New in version 0.23.0.

pandas.read_json

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]}'
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

to_latex(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=False*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires \usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

to_latex-specific options:

bold_rows [boolean, default False] Make the row labels bold in the output

column_format [str, default None] The columns format as specified in LaTeX table format e.g 'rcl' for 3 columns

longtable [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a \usepackage{longtable} to your LaTeX preamble.

escape [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

encoding [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

multicolumn [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module.

New in version 0.20.0.

multirow [boolean, default False] Use multirow to enhance MultiIndex rows. Requires adding a \usepackage{multirow} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

New in version 0.20.0.

to_msgpack (*path_or_buf=None*, *encoding='utf-8'*, ***kwargs*)
msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

path [string File path, buffer-like, or None] if None, return generated string

append [boolean whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

to_panel ()
Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Deprecated since version 0.20.0.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

panel : Panel

to_parquet (*fname*, *engine='auto'*, *compression='snappy'*, ***kwargs*)
Write a DataFrame to the binary parquet format.

New in version 0.21.0.

This function writes the dataframe as a parquet file. You can choose different parquet backends, and have the option of compression. See the user guide for more details.

fname [str] String file path.

engine [{'auto', 'pyarrow', 'fastparquet'}, default 'auto'] Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

compression [{'snappy', 'gzip', 'brotli', None}, default 'snappy'] Name of the compression to use. Use `None` for no compression.

****kwargs** Additional arguments passed to the parquet library. See pandas io for more details.

read_parquet : Read a parquet file. DataFrame.to_csv : Write a csv file. DataFrame.to_sql : Write to a sql table. DataFrame.to_hdf : Write to hdf.

This function requires either the fastparquet or pyarrow library.

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gzip', compression='gzip')
>>> pd.read_parquet('df.parquet.gzip')
   col1  col2
```

(continues on next page)

(continued from previous page)

```
0      1      3
1      2      4
```

to_period (*freq=None*, *axis=0*, *copy=True*)
Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

freq : string, default axis : {0 or 'index', 1 or 'columns'}, default 0

The axis to convert (the index by default)

copy [boolean, default True] If False then underlying input data is not copied

ts : TimeSeries with PeriodIndex

to_pickle (*path*, *compression='infer'*, *protocol=2*)
Pickle (serialize) object to file.

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

protocol [int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1]_ paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

read_pickle : Load pickled pandas object (or any object) from file. DataFrame.to_hdf : Write DataFrame to an HDF5 file. DataFrame.to_sql : Write DataFrame to a SQL database. DataFrame.to_parquet : Write a DataFrame to the binary parquet format.

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

to_records(*index=True*, *convert_datetime64=None*)

Convert DataFrame to a NumPy record array.

Index will be put in the 'index' field of the record array if requested.

index [boolean, default True] Include index in resulting record array, stored in 'index' field.

convert_datetime64 [boolean, default None] Deprecated since version 0.23.0.

Whether to convert the index to datetime.datetime if it is a DatetimeIndex.

y : numpy.recarray

DataFrame.from_records: convert structured or record ndarray to DataFrame.

numpy.recarray: ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   A     B
a  1  0.50
b  2  0.75
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

By default, timestamps are converted to *datetime.datetime*:

```
>>> df.index = pd.date_range('2018-01-01 09:00', periods=2, freq='min')
>>> df
                     A     B
2018-01-01 09:00:00  1  0.50
2018-01-01 09:01:00  2  0.75
>>> df.to_records()
rec.array([(datetime.datetime(2018, 1, 1, 9, 0), 1, 0.5 ),
           (datetime.datetime(2018, 1, 1, 9, 1), 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The timestamp conversion can be disabled so NumPy's datetime64 data type is used instead:

```
>>> df.to_records(convert_datetime64=False)
rec.array([('2018-01-01T09:00:00.000000000', 1, 0.5 ),
           ('2018-01-01T09:01:00.000000000', 2, 0.75)],
          dtype=[('index', '<M8[ns]'), ('A', '<i8'), ('B', '<f8')])
```

to_sparse(*fill_value=None*, *kind='block'*)

Convert to SparseDataFrame

fill_value : float, default NaN kind : {'block', 'integer'}

y : SparseDataFrame

to_sql(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1]_ are supported. Tables can be newly created, appended to, or overwritten.

name [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists [{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index [boolean, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize [int, optional] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

ValueError When the table already exists and *if_exists* is 'fail' (the default).

pandas.read_sql : read a DataFrame from a table

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
```

(continues on next page)

(continued from previous page)

```
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

to_stata(*fname*, *convert_dates=None*, *write_index=True*, *encoding='latin-1'*, *byteorder=None*, *time_stamp=None*, *data_label=None*, *variable_labels=None*, *version=114*, *convert_strl=None*)

Export Stata binary dta files.

fname [path (string), buffer or path object] string, path object (pathlib.Path or py._path.local.LocalPath) or object implementing a binary write() functions. If using a buffer then the buffer will not be automatically closed after the file data has been written.

convert_dates [dict] Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to 'tc'. Raises NotImplementedError if a datetime column has timezone information.

write_index [bool] Write the index to Stata dataset.

encoding [str] Default is latin-1. Unicode is not supported.

byteorder [str] Can be ">", "<", "little", or "big". default is *sys.byteorder*.

time_stamp [datetime] A datetime to use as file creation date. Default is the current time.

data_label [str] A label for the data set. Must be 80 characters or smaller.

variable_labels [dict] Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

New in version 0.19.0.

version [{114, 117}] Version to use in the output dta file. Version 114 can be used read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 114 limits string variables to 244 characters

or fewer while 117 allows strings with lengths up to 2,000,000 characters.

New in version 0.23.0.

convert_strl [list, optional] List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

New in version 0.23.0.

NotImplementedError

- If datetimes contain timezone information
- Column dtype is not representable in Stata

ValueError

- Columns listed in convert_dates are neither datetime64[ns] or datetime.datetime
- Column listed in convert_dates is not in DataFrame
- Categorical label contains more than 32,000 characters

New in version 0.19.0.

pandas.read_stata : Import Stata data files pandas.io.stata.StataWriter : low-level writer for Stata data files pandas.io.stata.StataWriter117 : low-level writer for version 117 files

```
>>> data.to_stata('./data_file.dta')
```

Or with dates

```
>>> data.to_stata('./date_data_file.dta', {2 : 'tw'})
```

Alternatively you can create an instance of the StataWriter class

```
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

With dates:

```
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
```

to_string(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *line_width=None*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*)

Render a DataFrame to a console-friendly tabular output.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_timestamp (*freq=None*, *how='start'*, *axis=0*, *copy=True*)
Cast to DatetimeIndex of timestamps, at *beginning* of period

freq [string, default frequency of PeriodIndex] Desired frequency

how [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default)

copy [boolean, default True] If false then underlying input data is not copied

df : DataFrame with DatetimeIndex

to_xarray()
Return an xarray object from the pandas object.

a DataArray for a Series a Dataset for a DataFrame a DataArray for higher dims

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
```

(continues on next page)

(continued from previous page)

```
   A    B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index    (index) int64 0 1 2
Data variables:
    A        (index) int64 1 1 2
    B        (index) object 'foo' 'bar' 'foo'
    C        (index) float64 4.0 5.0 6.0
```

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)}
                      ).set_index(['B','A'])
>>> df
         C
B   A
foo 1  4.0
bar 1  5.0
foo 2  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B        (B) object 'bar' 'foo'
  * A        (A) int64 1 2
Data variables:
    C        (B, A) float64 5.0 nan 4.0 6.0
```

```
>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                 items=list('ABCD'),
                 major_axis=pd.date_range('20130101', periods=3),
                 minor_axis=['first', 'second'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second
```

```
>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],
       [[ 6,  7],
        [ 8,  9],
        [10, 11]],
       [[12, 13],
```

(continues on next page)

(continued from previous page)

```
        [14, 15],
        [16, 17]],
       [[18, 19],
        [20, 21],
        [22, 23]]])
Coordinates:
  * items       (items) object 'A' 'B' 'C' 'D'
  * major_axis  (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03
↪# noqa
  * minor_axis  (minor_axis) object 'first' 'second'
```

See the xarray docs

transform(*func*, **args*, ***kwargs*)

Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values

New in version 0.20.0.

func [callable, string, dictionary, or list of string/callables] To apply to column

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

transformed : NDFrame

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
...                   index=pd.date_range('1/1/2000', periods=10))
df.iloc[3:7] = np.nan
```

```
>>> df.transform(lambda x: (x - x.mean()) / x.std())
                   A         B         C
2000-01-01  0.579457  1.236184  0.123424
2000-01-02  0.370357 -0.605875 -1.231325
2000-01-03  1.455756 -0.277446  0.288967
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -0.498658  1.274522  1.642524
2000-01-09 -0.540524 -1.012676 -0.828968
2000-01-10 -1.366388 -0.614710  0.005378
```

pandas.NDFrame.aggregate pandas.NDFrame.apply

transpose(**args*, ***kwargs*)

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
                0     1
name        Alice   Bob
score         9.5     8
employed    False  True
kids            0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

truediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

before [date, string, int] Truncate all rows before this index value.

after [date, string, int] Truncate all rows after this index value.

axis [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

copy [boolean, default is True,] Return a copy of the truncated section.

type of caller The truncated Series or DataFrame.

DataFrame.loc : Select a subset of a DataFrame by label. DataFrame.iloc : Select a subset of a DataFrame by position.

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
```

(continues on next page)

(continued from previous page)

```
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

tshift (*periods=1*, *freq=None*, *axis=0*)
Shift the time index, using the index's frequency if available.

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

shifted : NDFrame

tz_convert (*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

tz : string or pytz.timezone object axis : the axis to convert level : int, str, default None

If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

TypeError If the axis is tz-naive.

tz_localize (*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*)
Localize tz-naive TimeSeries to target time zone.

tz : string or pytz.timezone object axis : the axis to localize level : int, str, default None

If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise']

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times

- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

TypeError If the TimeSeries is tz-aware and tz is not None.

unstack (*level=-1*, *fill_value=None*)

Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels. If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex). The level involved will automatically get sorted.

level [int, string, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name

fill_value [replace NaN with this value if the unstack produces] missing values

New in version 0.18.0.

DataFrame.pivot : Pivot a table based on column values. DataFrame.stack : Pivot a level of the column labels (inverse operation

from *unstack*).

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                    ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a   1.0
     b   2.0
two  a   3.0
     b   4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a   b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
   one  two
a  1.0   3.0
b  2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a  1.0
     b  2.0
two  a  3.0
     b  4.0
dtype: float64
```

unstacked : DataFrame or Series

update (*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

other [DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

join [{'left'}, default 'left'] Only left join is implemented, keeping the index and columns of the original object.

overwrite [bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.
- False: only update values that are NA in the original DataFrame.

filter_func [callable(1d-array) -> boolean 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

raise_conflict [bool, default False] If True, will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

ValueError When *raise_conflict* is True and there's overlapping non-NA data.

dict.update : Similar method for dictionaries. DataFrame.merge : For column(s)-on-columns(s) operations.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
1  b  e
2  c  f
```

For Series, it's name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
```

(continues on next page)

(continued from previous page)

```
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A      B
0  1    4.0
1  2  500.0
2  3    6.0
```

values

Return a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

numpy.ndarray The values of the DataFrame.

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age':    [ 3,  29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3      94      31
1   29     170     115
>>> df.dtypes
age       int64
height    int64
weight    int64
dtype: object
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('monkey', np.nan, None)],
...                    columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name          object
max_speed    float64
rank          object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
```

(continues on next page)

(continued from previous page)

```
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

pandas.DataFrame.index : Retrievie the index labels pandas.DataFrame.columns : Retrieving the column names

var(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

var : Series or DataFrame (if level specified)

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
```

(continues on next page)

(continued from previous page)

```
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

DataFrame.mask()

xs(*key*, *axis=0*, *level=None*, *drop_level=True*)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on the rows (axis=0).

key [object] Some label contained in the index, or partially in a MultiIndex

axis [int, default 0] Axis to retrieve cross-section on

level [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level [boolean, default True] If False, returns object with same levels as self.

```
>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
A    4
B    5
C    2
Name: a
>>> df.xs('C', axis=1)
a    2
b    9
c    3
Name: C
```

```
>>> df
                    A  B  C  D
first second third
bar   one    1      4  1  8  9
      two    1      7  5  5  0
baz   one    1      6  6  8  0
      three  2      5  3  5  3
>>> df.xs(('baz', 'three'))
       A  B  C  D
third
2      5  3  5  3
>>> df.xs('one', level=1)
             A  B  C  D
first third
bar   1      4  1  8  9
baz   1      6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
        A  B  C  D
```

(continues on next page)

(continued from previous page)

```
second
three    5  3  5  3
```

xs : Series or DataFrame

xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of xs functionality, see MultiIndex Slicers

class Fred2.Core.Result.**CleavageFragmentPredictionResult**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

Bases: *Fred2.Core.Result.AResult*

A *CleavageFragmentPredictionResult* object is a `pandas.DataFrame` with single-indexing, where column Ids are the prediction scores fo the different prediction methods, and row ID the *Peptide* object.

CleavageFragmentPredictionResult:

Peptide Obj	Method Name
Peptide1	-15.34
Peptide2	23.34

T

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
                0     1
name        Alice   Bob
score         9.5     8
employed    False  True
kids            0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name          object
score        float64
employed        bool
kids           int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

abs()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

abs Series/DataFrame containing the absolute value of each element.

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
```

(continues on next page)

(continued from previous page)

```
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
     a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
     a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

numpy.absolute : calculate the absolute value element-wise.

add(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.radd

add_prefix(*prefix*)

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

prefix [str] The string to add before each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_suffix: Suffix row labels with string *suffix*. DataFrame.add_suffix: Suffix column labels with string *suffix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
```

(continues on next page)

(continued from previous page)

```
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0       1       3
1       2       4
2       3       5
3       4       6
```

add_suffix(*suffix*)

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

suffix [str] The string to add after each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_prefix: Prefix row labels with string *prefix*. DataFrame.add_prefix: Prefix column labels with string *prefix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

agg (*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

aggregate (*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

align(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)

Align two objects on their axes with the specified join method for each axis Index

other : DataFrame or Series join : {'outer', 'inner', 'left', 'right'}, default 'outer' axis : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None)

level [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level

copy [boolean, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

method : str, default None limit : int, default None fill_axis : {0 or 'index', 1 or 'columns'}, default 0

Filling axis, method and limit

broadcast_axis [{0 or 'index', 1 or 'columns'}, default None] Broadcast values along this axis, if aligning two objects of different dimensions

(left, right) [(DataFrame, type of other)] Aligned objects

all (*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether all elements are True, potentially over an axis.

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

all : Series or DataFrame (if level specified)

pandas.Series.all : Return True if all elements are True pandas.DataFrame.any : Return True if one (or more) elements are True

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

any(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

any : Series or DataFrame (if level specified)

pandas.DataFrame.all : Return whether all elements are True.

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([True, False]).any()
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
```

(continues on next page)

(continued from previous page)

```
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

append(*other*, *ignore_index=False*, *verify_integrity=False*, *sort=None*)

Append rows of *other* to the end of this frame, returning a new object. Columns not in this frame are added as new columns.

other [DataFrame or Series/dict-like object, or list of these] The data to append.

ignore_index [boolean, default False] If True, do not use the index labels.

verify_integrity [boolean, default False] If True, raise ValueError on creating index with duplicates.

sort [boolean, default None] Sort columns if the columns of *self* and *other* are not aligned. The default sorting is deprecated and will change to not-sorting in a future version of pandas. Explicitly pass `sort=True` to silence the warning and sort. Explicitly pass `sort=False` to silence the warning and not sort.

New in version 0.23.0.

appended : DataFrame

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

pandas.concat [General function to concatenate DataFrame, Series] or Panel objects

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With *ignore_index* set to True:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
...     df = df.append({'A': i}, ignore_index=True)
>>> df
   A
0  0
1  1
2  2
3  3
4  4
```

More efficient:

```
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
...           ignore_index=True)
   A
0  0
1  1
2  2
3  3
4  4
```

apply (*func*, *axis=0*, *broadcast=None*, *raw=False*, *reduce=None*, *result_type=None*, *args=()*, ***kwds*)
Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

func [function] Function to apply to each column or row.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

broadcast [bool, optional] Only relevant for aggregation functions:

- `False` or `None` : returns a Series whose length is the length of the index or the number of columns (based on the *axis* parameter)
- `True` : results will be broadcast to the original shape of the frame, the original index and columns will be retained.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by result_type='broadcast'.

raw [bool, default False]

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

reduce [bool or None, default None] Try to apply reduction procedures. If the DataFrame is empty, *apply* will use *reduce* to determine whether the result should be a Series or a DataFrame. If `reduce=None` (the default), *apply*'s return value will be guessed by calling *func* on an empty Series (note: while guessing, exceptions raised by *func* will be ignored). If `reduce=True` a Series will always be returned, and if `reduce=False` a DataFrame will always be returned.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by `result_type='reduce'`.

result_type [{'expand', 'reduce', 'broadcast', None}, default None] These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.
- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

New in version 0.23.0.

args [tuple] Positional arguments to pass to *func* in addition to the array/series.

****kwds** Additional keyword arguments to pass as keywords arguments to *func*.

In the current implementation apply calls *func* twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if *func* has side-effects, as they will take effect twice for the first column/row.

DataFrame.applymap: For elementwise operations DataFrame.aggregate: only perform aggregating type operations DataFrame.transform: only perform transformating type operations

```
>>> df = pd.DataFrame([[4, 9],] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
```

(continues on next page)

(continued from previous page)

```
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Retuning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing result_type='expand' will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
```

(continues on next page)

(continued from previous page)

```
1  1  2
2  1  2
```

applied : Series or DataFrame

applymap (*func*)

Apply a function to a Dataframe elementwise.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

func [callable] Python function, returns a single value from a single value.

DataFrame Transformed DataFrame.

DataFrame.apply : Apply a function along input axis of DataFrame

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.applymap(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.applymap(lambda x: x**2)
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

But it's better to avoid applymap in that case.

```
>>> df ** 2
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

as_blocks (*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

copy : boolean, default True

values : a dict of dtype -> Constructor Types

as_matrix (*columns=None*)

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

columns: list, optional, default:None If None, return all columns, otherwise, returns specified columns.

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a flot64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

pandas.DataFrame.values

asfreq(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)
Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

freq : DateOffset object, or string method : {'backfill'/'bfill', 'pad'/'ffill'}, default None

Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill

how [{'start', 'end'}, default end] For PeriodIndex only, see PeriodIndex.asfreq

normalize [bool, default False] Whether to reset output index to midnight

fill_value: scalar, optional Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

converted : type of caller

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:03:00    3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                     s
2000-01-01 00:00:00    0.0
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    2.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    3.0
2000-01-01 00:03:00    3.0
```

reindex

To learn more about the frequency strings, please see this link.

asof (*where*, *subset=None*)

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

where : date or array of dates subset : string or list of strings, default None

if not None use these columns for NaN propagation

Dates are assumed to be sorted Raises if this is not the case

where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

where is Index: same shape object as input

merge_asof

assign (***kwargs*)

Assign new columns to a DataFrame, returning a new object (a copy) with the new columns added to the original ones. Existing columns that are re-assigned will be overwritten.

kwargs [keyword, value pairs] keywords are the column names. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

df [DataFrame] A new DataFrame with the new columns in addition to all the existing columns.

Assigning multiple columns within the same `assign` is possible. For Python 3.6 and above, later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order. For Python 3.5 and below, the order of keyword arguments is not specified, you cannot refer to newly created or modified columns. All items are computed first, and then assigned in alphabetical order.

Changed in version 0.23.0: Keyword argument order is maintained for Python 3.6 and later.

```
>>> df = pd.DataFrame({'A': range(1, 11), 'B': np.random.randn(10)})
```

Where the value is a callable, evaluated on *df*:

```
>>> df.assign(ln_A = lambda x: np.log(x.A))
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the value already exists and is inserted:

```
>>> newcol = np.log(df['A'])
>>> df.assign(ln_A=newcol)
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the keyword arguments depend on each other

```
>>> df = pd.DataFrame({'A': [1, 2, 3]})
```

```
>>> df.assign(B=df.A, C=lambda x:x['A']+ x['B'])
    A  B  C
 0  1  1  2
 1  2  2  4
 2  3  3  6
```

astype(***kwargs*)

Cast a pandas object to a specified dtype `dtype`.

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy [bool, default True.] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors [{'raise', 'ignore'}, default 'raise'.] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

raise_on_error [raise on invalid input] Deprecated since version 0.20.0: Use `errors` instead

kwargs : keyword arguments to pass on to the constructor

casted : type of caller

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> ser.astype('category', ordered=True, categories=[2, 1])
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to a numeric type. numpy.ndarray.astype : Cast a numpy array to a specified type.

at

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.iat [Access a single value for a row/column pair by integer] position

DataFrame.loc : Access a group of rows and columns by label(s) Series.at : Access a single value using a label

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

KeyError When label does not exist in DataFrame

at_time (*time*, *asof=False*)

Select values at particular time of day (e.g. 9:30AM).

TypeError If the index is not a `DatetimeIndex`

time : datetime.time or string

values_at_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

between_time : Select values between particular times of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_at_time : Get just the index locations for

values at particular time of the day

axes

Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['col1', 'col2'],
dtype='object')]
```

between_time (*start_time*, *end_time*, *include_start=True*, *include_end=True*)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

TypeError If the index is not a `DatetimeIndex`

start_time : datetime.time or string end_time : datetime.time or string include_start : boolean, default True include_end : boolean, default True

values_between_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

at_time : Select values at a particular time of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_between_time : Get just the index locations for

values between particular times of the day

bfill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='bfill')`

blocks
Internal property, property synonym for as_blocks()

Deprecated since version 0.21.0.

bool()
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

boxplot(*column=None*, *by=None*, *ax=None*, *fontsize=None*, *rot=0*, *grid=True*, *figsize=None*, *layout=None*, *return_type=None*, ***kwds*)
Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to *1.5 * IQR (IQR = Q3 - Q1)* from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia's entry for boxplot.

column [str or list of str, optional] Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.

by [str or array-like, optional] Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in *by*.

ax [object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.

fontsize [float or str] Tick label font size in points or as a string (e.g., *large*).

rot [int or float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate sytem.

grid [boolean, default True] Setting this to True will show the grid.

figsize [A tuple (width, height) in inches] The size of the figure to create in matplotlib.

layout [tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 columns and 5 rows, starting from the top-left.

return_type [{'axes', 'dict', 'both'} or None, default 'axes'] The kind of object to return. The default is `axes`.

- 'axes' returns the matplotlib axes the boxplot is drawn on.
- 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot.
- 'both' returns a namedtuple with the axes and dict.
- when grouping with `by`, a Series mapping columns to `return_type` is returned.

  If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

result :

The return type depends on the *return_type* parameter:

- 'axes' : object of class matplotlib.axes.Axes

- 'dict' : dict of matplotlib.lines.Line2D objects
- 'both' : a nametuple with strucure (ax, lines)

For data grouped with `by`:

- `Series`
- `array` (for `return_type = None`)

Series.plot.hist: Make a histogram. matplotlib.pyplot.boxplot : Matplotlib equivalent plot.

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`. For instance:

A list of strings (i.e. `['X', 'Y']`) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

The layout of boxplot can be adjusted giving a tuple to `layout`:

Additional formatting can be done to the boxplot, like suppressing the grid (`grid=False`), rotating the labels in the x-axis (i.e. `rot=45`) or changing the fontsize (i.e. `fontsize=15`):

The parameter `return_type` can be used to select the type of element returned by *boxplot*. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```
>>> boxplot = df.boxplot(column=['Col1','Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

When grouping with `by`, a Series mapping columns to `return_type` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned:

```
>>> boxplot =  df.boxplot(column=['Col1', 'Col2'], by='X',
...                       return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

clip (*lower=None*, *upper=None*, *axis=None*, *inplace=False*, **args*, ***kwargs*)
Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

lower [float or array_like, default None] Minimum threshold value. All values below this threshold will be set to it.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

clip_lower : Clip values below specified threshold(s). clip_upper : Clip values above specified threshold(s).

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

clip_lower(*threshold*, *axis=None*, *inplace=False*)

Return copy of the input with values below a threshold truncated.

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Series.clip [Return copy of input with values below and above] thresholds truncated.

Series.clip_upper [Return copy of input with values above] threshold truncated.

clipped : same type as input

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

clip_upper (*threshold*, *axis=None*, *inplace=False*)

Return copy of input with values above given value(s) truncated.

threshold : float or array_like axis : int or string axis name, optional

Align object with threshold along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data

New in version 0.21.0.

clip

clipped : same type as input

columns

The column labels of the DataFrame.

combine (*other*, *func*, *fill_value=None*, *overwrite=True*)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

other : DataFrame func : function

Function that takes two series as inputs and return a Series or a scalar

fill_value : scalar value overwrite : boolean, default True

If True then overwrite values for common keys in the calling frame

result : DataFrame

```
>>> df1 = DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, lambda s1, s2: s1 if s1.sum() < s2.sum() else s2)
   A  B
0  0  3
1  0  3
```

DataFrame.combine_first [Combine two DataFrame objects and default to] non-null values in frame calling the method

combine_first (*other*)

Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

other : DataFrame

combined : DataFrame

df1's values prioritized, use values from df2 to fill holes:

```
>>> df1 = pd.DataFrame([[1, np.nan]])
>>> df2 = pd.DataFrame([[3, 4]])
>>> df1.combine_first(df2)
   0    1
0  1  4.0
```

DataFrame.combine [Perform series-wise operation on two DataFrames] using a given function

compound(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

compounded : Series or DataFrame (if level specified)

consolidate(*inplace=False*)

Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

convert_objects(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Return a fixed frequency timedelta index,

with day as the default.

converted : same as input object

copy(*deep=True*)

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

deep [bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

copy [Series, DataFrame or Panel] Object type matches caller.

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
```

(continues on next page)

(continued from previous page)

```
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

corr(*method='pearson'*, *min_periods=1*)

Compute pairwise correlation of columns, excluding NA/null values

method [{'pearson', 'kendall', 'spearman'}]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation

y : DataFrame

corrwith(*other*, *axis=0*, *drop=False*)

Compute pairwise correlation between rows or columns of two DataFrame objects.

other : DataFrame, Series axis : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' to compute column-wise, 1 or 'columns' for row-wise

drop [boolean, default False] Drop missing indices from result, default returns union of all

correls : Series

count(*axis=0*, *level=None*, *numeric_only=False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, and optionally *numpy.inf* (depending on *pandas.options.mode.use_inf_as_na*) are considered NA.

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each **row**.

level [int or str, optional] If the axis is a *MultiIndex* (hierarchical), count along a particular *level*, collapsing into a *DataFrame*. A *str* specifies the level name.

numeric_only [boolean, default False] Include only *float*, *int* or *boolean* data.

Series or DataFrame For each column/row the number of non-NA/null entries. If *level* is specified returns a *DataFrame*.

Series.count: number of non-NA elements in a Series DataFrame.shape: number of DataFrame rows and columns (including NA

elements)

DataFrame.isna: boolean same-sized DataFrame showing places of NA elements

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                    ["John", "Myla", None, "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> df
   Person   Age  Single
0    John  24.0   False
1    Myla   NaN    True
2    None  21.0    True
3    John  33.0    True
4    Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    4
Age       4
Single    5
dtype: int64
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    2
3    3
4    3
dtype: int64
```

Counts for one level of a *MultiIndex*:

```
>>> df.set_index(["Person", "Single"]).count(level="Person")
        Age
Person
John      2
Myla      1
```

cov (*min_periods=None*)

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as `NaN`.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result.

DataFrame The covariance matrix of the series of the DataFrame.

pandas.Series.cov : compute covariance with another Series pandas.core.window.EWM.cov: expoential weighted sample covariance pandas.core.window.Expanding.cov : expanding sample covariance pandas.core.window.Rolling.cov : rolling sample covariance

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                   columns=['dogs', 'cats'])
>>> df.cov()
          dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
          a         b         c         d         e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
```

(continues on next page)

(continued from previous page)

```
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
          a         b         c
a  0.316741       NaN -0.150812
b       NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

cummax (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummax : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.max [Similar functionality] but ignores `NaN` values.

DataFrame.max [Return the maximum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cummin (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummin : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
```

(continues on next page)

(continued from previous page)

```
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.min [Similar functionality] but ignores `NaN` values.

DataFrame.min [Return the minimum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumprod (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumprod : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.prod [Similar functionality] but ignores `NaN` values.

DataFrame.prod [Return the product over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumsum(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumsum : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.sum [Similar functionality] but ignores `NaN` values.

DataFrame.sum [Return the sum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

describe (*percentiles=None*, *include=None*, *exclude=None*)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

summary: Series/DataFrame of summary statistics

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                       3
unique                      2
top       2010-01-01 00:00:00
freq                        2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                     'numeric': [1, 2, 3],
...                     'categorical': pd.Categorical(['d','e','f'])
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
        categorical  numeric object
count             3      3.0      3
unique            3      NaN      3
top               f      NaN      c
freq              1      NaN      1
mean            NaN      2.0    NaN
std             NaN      1.0    NaN
min             NaN      1.0    NaN
25%             NaN      1.5    NaN
50%             NaN      2.0    NaN
75%             NaN      2.5    NaN
max             NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
        categorical  numeric
count            3      3.0
```

(continues on next page)

(continued from previous page)

```
unique            3      NaN
top               f      NaN
freq              1      NaN
mean            NaN      2.0
std             NaN      1.0
min             NaN      1.0
25%             NaN      1.5
50%             NaN      2.0
75%             NaN      2.5
max             NaN      3.0
```

DataFrame.count DataFrame.max DataFrame.min DataFrame.mean DataFrame.std DataFrame.select_dtypes

diff (*periods=1*, *axis=0*)

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

periods [int, default 1] Periods to shift for calculating difference, accepts negative values.

axis [{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

New in version 0.16.1..

diffed : DataFrame

Series.diff: First discrete difference for a Series. DataFrame.pct_change: Percent change over given number of periods. DataFrame.shift: Shift index by desired number of periods with an

optional time freq.

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a    b     c
0 NaN  0.0   0.0
1 NaN -1.0   3.0
2 NaN -1.0   7.0
3 NaN -1.0  13.0
4 NaN  0.0  20.0
5 NaN  2.0  28.0
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

div(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

divide(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

dot (*other*)

Matrix multiplication with DataFrame or Series objects. Can also be called using *self @ other* in Python >= 3.5.

other : DataFrame or Series

dot_product : DataFrame or Series

drop (*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

labels [single label or list-like] Index or column labels to drop.

axis [{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index, columns [single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

New in version 0.21.0.

level [int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace [bool, default False] If True, do operation inplace and return None.

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

dropped : pandas.DataFrame

DataFrame.loc : Label-location based indexer for selection by label. DataFrame.dropna : Return DataFrame with labels on given axis omitted

where (all or any) data are missing

DataFrame.drop_duplicates [Return DataFrame with duplicate rows] removed, optionally only considering certain columns

Series.drop : Return Series with specified index labels removed.

KeyError If none of the labels are found in the selected axis

```
>>> df = pd.DataFrame(np.arange(12).reshape(3,4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                              [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3,0.2]])
>>> df
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
lama    speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
```

(continues on next page)

(continued from previous page)

```
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

drop_duplicates (*subset=None*, *keep='first'*, *inplace=False*)

Return DataFrame with duplicate rows removed, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Drop duplicates except for the first occurrence.
- `last` : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

inplace [boolean, default False] Whether to drop duplicates in place or to return a copy

deduplicated : DataFrame

dropna (*axis=0*, *how='any'*, *thresh=None*, *subset=None*, *inplace=False*)

Remove missing values.

See the User Guide for more on which values are considered missing, and how to work with missing data.

axis [{0 or 'index', 1 or 'columns'}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Deprecated since version 0.23.0:: Pass tuple or list to drop on multiple

axes.

how [{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

thresh [int, optional] Require that many non-NA values.

subset [array-like, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace [bool, default False] If True, do operation inplace and return None.

DataFrame DataFrame with NA entries dropped from it.

DataFrame.isna: Indicate missing values. DataFrame.notna : Indicate existing (non-missing) values. DataFrame.fillna : Replace missing values. Series.dropna : Drop missing values. Index.dropna : Drop missing indices.

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'born'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type of each column.

pandas.DataFrame.ftypes : dtype and sparsity information.

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
string              object
dtype: object
```

duplicated(*subset=None*, *keep='first'*)

Return boolean Series denoting duplicate rows, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Mark duplicates as `True` except for the first occurrence.
- `last` : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

duplicated : Series

empty

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

bool If DataFrame is empty, return True, if not return False.

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
```

(continues on next page)

(continued from previous page)

```
>>> df.dropna().empty
True
```

pandas.Series.dropna pandas.DataFrame.dropna

eq(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods eq

equals(*other*)
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

eval(*expr*, *inplace=False*, ***kwargs*)
Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

expr [str] The expression string to evaluate.

inplace [bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

New in version 0.18.0..

kwargs [dict] See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

ndarray, scalar, or pandas object The result of the evaluation.

DataFrame.query [Evaluates a boolean expression to query the columns] of a frame.

DataFrame.assign [Can evaluate an expression or function to create new] values for a column.

pandas.eval [Evaluate a Python expression as a string using various] backends.

For more details see the API documentation for `eval()`. For detailed examples see enhancing performance with eval.

```
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
>>> df.eval('A + B')
0    11
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
```

Use `inplace=True` to modify the original DataFrame.

```
>>> df.eval('C = A + B', inplace=True)
>>> df
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
```

ewm(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*)
Provides exponential weighted functions

New in version 0.18.0.

com [float, optional] Specify decay in terms of center of mass, $\alpha = 1/(1 + com)$, for $com \geq 0$

span [float, optional] Specify decay in terms of span, $\alpha = 2/(span + 1)$, for $span \geq 1$

halflife [float, optional] Specify decay in terms of half-life, $\alpha = 1 - exp(log(0.5)/halflife)$, for $halflife > 0$

alpha [float, optional] Specify smoothing factor α directly, $0 < \alpha \leq 1$

New in version 0.18.0.

min_periods [int, default 0] Minimum number of observations in window required to have a value (otherwise result is NA).

adjust [boolean, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

ignore_na [boolean, default False] Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), . . . , 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

rolling : Provides rolling window calculations expanding : Provides expanding transformations.

expanding (*min_periods=1*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

min_periods [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

center [boolean, default False] Set the labels at the center of the window.

axis : int or string, default 0

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

rolling : Provides rolling window calculations ewm : Provides exponential weighted functions

ffill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='ffill')`

fillna(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)
Fill NA/NaN values using the specified method

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis : {0 or 'index', 1 or 'columns'} inplace : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

interpolate : Fill NaN values using interpolation. reindex, asfreq

filled : DataFrame

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 2.0 1
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

filter (*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

items [list-like] List of info axis to restrict to (must not all be present)

like [string] Keep info axis where "arg in col == True"

regex [string (regular expression)] Keep info axis with re.search(regex, col) == True

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame

same type as input object

```
>>> df
one  two  three
mouse     1    2      3
rabbit    4    5      6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
one  two  three
rabbit    4    5      6
```

pandas.DataFrame.loc

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

filter_result(*expressions*)

Filters a result data frame based on a specified expression consisting of a list of triple with (method_name, comparator, threshold). The expression is applied to each row. If any of the columns fulfill the criteria the row remains.

Parameters **expressions** (*list((str, comparator, float))*) – A list of triples consisting of (method_name, comparator, threshold)

Returns A new filtered result object

Return type *CleavageFragmentPredictionResult*

first(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

subset : type of caller

last : Select final periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

first_valid_index()

Return index for first non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

floordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rfloordiv

classmethod from_csv(*path*, *header=0*, *sep=','*, *','*, *index_col=0*, *parse_dates=True*, *encoding=None*, *tupleize_cols=None*, *infer_datetime_format=False*)
Read CSV file.

Deprecated since version 0.21.0: Use `pandas.read_csv()` instead.

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a DataFrame of time series data.

This method only differs from the preferred `pandas.read_csv()` in some defaults:

- *index_col* is `0` instead of `None` (take first column as index by default)
- *parse_dates* is `True` instead of `False` (try parsing the index as datetime by default)

So a `pd.DataFrame.from_csv(path)` can be replaced by `pd.read_csv(path, index_col=0, parse_dates=True)`.

path : string file path or file handle / StringIO header : int, default 0

Row to use as header (skip prior rows)

sep [string, default ','] Field delimiter

index_col [int or sequence, default 0] Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

parse_dates [boolean, default True] Parse dates. Different default from read_table

tupleize_cols [boolean, default False] write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

infer_datetime_format: boolean, default False If True and *parse_dates* is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

pandas.read_csv

y : DataFrame

classmethod from_dict(*data*, *orient='columns'*, *dtype=None*, *columns=None*)
Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

data [dict] Of the form {field : array-like} or {field : dict}.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'.

dtype [dtype, default None] Data type to force, otherwise infer.

columns [list, default None] Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'`.

New in version 0.23.0.

pandas.DataFrame

DataFrame.from_records [DataFrame from ndarray (structured] dtype), list of tuples, dict, or DataFrame

DataFrame : DataFrame object creation using constructor

By default the keys of the dict become the DataFrame columns:

```
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Specify `orient='index'` to create the DataFrame using dictionary keys as rows:

```
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
       0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                        columns=['A', 'B', 'C', 'D'])
       A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
```

classmethod from_items(*items*, *columns=None*, *orient='columns'*)
Construct a dataframe from a list of tuples

Deprecated since version 0.23.0: *from_items* is deprecated and will be removed in a future version. Use `DataFrame.from_dict(dict(items))` instead. `DataFrame.from_dict(OrderedDict(items))` may be used to preserve the key order.

Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the columns, but depends on the specified orientation). The values should be arrays or Series.

items [sequence of (key, value) pairs] Values should be arrays or Series.

columns [sequence of column labels, optional] Must be passed if orient='index'.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the input correspond to column labels, pass 'columns' (default). Otherwise if the keys correspond to the index, pass 'index'.

frame : DataFrame

classmethod from_records(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float=False*, *nrows=None*)

Convert structured or record ndarray to DataFrame

data : ndarray (structured dtype), list of tuples, dict, or DataFrame index : string, list of fields, array-like

Field of array to use as the index, alternately a specific set of input labels to use

exclude [sequence, default None] Columns or fields to exclude

columns [sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

coerce_float [boolean, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

df : DataFrame

ftypes

Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type and indication of sparse/dense of each column.

pandas.DataFrame.dtypes: Series with just dtype information. pandas.SparseDataFrame : Container for sparse tabular data.

Sparse data should have the same dtypes as its dense representation.

```
>>> import numpy as np
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).ftypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).ftypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

ge(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods ge

get(*key*, *default=None*)
Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

key : object

value : type of items contained in object

get_dtype_counts()
Return counts of unique dtypes in this object.

dtype [Series] Series with the count of columns with each dtype.

dtypes : Return the dtypes in this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

get_ftype_counts()
Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

ftypes [Return ftypes (indication of sparse/dense and dtype) in] this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

get_value(*index*, *col*, *takeable=False*)
Quickly retrieve single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label takeable : interpret the index/col as indexers, default False

value : scalar value

get_values()

Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a *pandas.SparseArray*, the data are first converted to a dense representation.

numpy.ndarray Numpy representation of DataFrame

values : Numpy representation of DataFrame. pandas.SparseArray : Container for sparse data.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a      b    c
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
     a    c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

groupby(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=False*, *observed=False*, ***kwargs*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

by [mapping, function, label, or list of labels] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted a (single) key.

axis : int, default 0 level : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

as_index [boolean, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort [boolean, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.

group_keys [boolean, default True] When calling apply, add group keys to index to identify pieces

squeeze [boolean, default False] reduce the dimensionality of the return type if possible, otherwise return a consistent type

observed [boolean, default False] This only applies if any of the groupers are Categoricals If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

GroupBy object

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

See the user guide for more.

resample [Convenience method for frequency conversion and resampling] of time series.

gt (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods gt

head (*n=5*)
Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

n [int, default 5] Number of rows to select.

obj_head [type of caller] The first *n* rows of the caller object.

pandas.DataFrame.tail: Returns the last *n* rows.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
```

(continues on next page)

(continued from previous page)

```
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

hist(*column=None*, *by=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *ax=None*, *sharex=False*, *sharey=False*, *figsize=None*, *layout=None*, *bins=10*, ***kwds*)
Make a histogram of the DataFrame's.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

data [DataFrame] The pandas object holding the data.

column [string or sequence] If passed, will be used to limit data to a subset of columns.

by [object, optional] If passed, then used to form histograms for separate groups.

grid [boolean, default True] Whether to show axis grid lines.

xlabelsize [int, default None] If specified changes the x-axis label size.

xrot [float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize [int, default None] If specified changes the y-axis label size.

yrot [float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

ax [Matplotlib axes object, default None] The axes to plot the histogram on.

sharex [boolean, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey [boolean, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize [tuple] The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

layout [tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

axes : matplotlib.AxesSubplot or numpy.ndarray of them

matplotlib.pyplot.hist : Plot a histogram using matplotlib.

iat

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.at : Access a single value for a row/column label pair DataFrame.loc : Access a group of rows and columns by label(s) DataFrame.iloc : Access a group of rows and columns by integer position(s)

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

IndexError When integer position is out of bounds

idxmax (*axis=0*, *skipna=True*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmax : Series

This method is the DataFrame version of `ndarray.argmax`.

Series.idxmax

idxmin (*axis=0*, *skipna=True*)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmin : Series

This method is the DataFrame version of `ndarray.argmin`.

Series.idxmin

iloc

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at Selection by Position

index

The index (row labels) of the DataFrame.

infer_objects()

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to numeric typeR

converted : same type as input object

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

info(*verbose=None*, *buf=None*, *max_cols=None*, *memory_usage=None*, *null_counts=None*)
Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

verbose [bool, optional] Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

buf [writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

max_cols [int, optional] When to switch from the verbose to the truncated output. If the DataFrame has more than *max_cols* columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

memory_usage [bool, str, optional] Specifies whether total memory usage of the DataFrame elements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources.

null_counts [bool, optional] Whether to show the non-null counts. By default, this is shown only if the frame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

None This method prints a summary of a DataFrame and returns None.

DataFrame.describe: Generate descriptive statistics of DataFrame columns.

DataFrame.memory_usage: Memory usage of DataFrame columns.

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                    "float_col": float_values})
>>> df
   int_col text_col  float_col
0        1    alpha       0.00
1        2     beta       0.25
2        3    gamma       0.50
3        4    delta       0.75
4        5  epsilon       1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
int_col      5 non-null int64
```

(continues on next page)

(continued from previous page)

```
text_col     5 non-null object
float_col    5 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w", encoding="utf-8") as f:
...     f.write(s)
260
```

The *memory_usage* parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 22.9+ MB
```

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 188.8 MB
```

insert (*loc*, *column*, *value*, *allow_duplicates=False*)

Insert column into DataFrame at specified location.

Raises a ValueError if *column* is already contained in the DataFrame, unless *allow_duplicates* is set to

True.

loc [int] Insertion index. Must verify 0 <= loc <= len(columns)

column [string, number, or hashable object] label of the inserted column

value : int, Series, or array-like allow_duplicates : bool, optional

interpolate(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

method [{'linear', 'time', 'index', 'values', 'nearest', 'zero',]

'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to `scipy.interpolate.interp1d`. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
- 'from_derivatives' refers to BPoly.from_derivatives which replaces 'piecewise_polynomial' interpolation method in scipy 0.18

New in version 0.18.1: Added support for the 'akima' method Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in scipy 0.18; backwards-compatible with scipy < 0.18

axis [{0, 1}, default 0]

- 0: fill column-by-column
- 1: fill row-by-row

limit [int, default None.] Maximum number of consecutive NaNs to fill. Must be greater than 0.

limit_direction : {'forward', 'backward', 'both'}, default 'forward' limit_area : {'inside', 'outside'}, default None

- None: (default) no fill restriction
- 'inside' Only fill NaNs surrounded by valid values (interpolate).
- 'outside' Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

inplace [bool, default False] Update the NDFrame in place if possible.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

kwargs : keyword arguments to pass on to the interpolating function.

Series or DataFrame of same shape interpolated at the NaNs

reindex, replace, fillna

Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

is_copy

isin(*values*)

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

values [iterable, Series, DataFrame or dictionary] The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dictionary, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

DataFrame of booleans

When `values` is a list:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
       A      B
0   True   True
1  False  False
2   True  False
```

When `values` is a dict:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
       A      B
0   True  False  # Note that B didn't match the 1 here.
1  False   True
2   True   True
```

When `values` is a Series or DataFrame:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
       A      B
0   True  False
1  False  False  # Column A in `other` has a 3, but not at index 1.
2   True   True
```

isna()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

isnull()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

items()

Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iteritems()

Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iterrows()

Iterate over DataFrame rows as (index, Series) pairs.

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
```

(continues on next page)

(continued from previous page)

```
>>> print(df['int'].dtype)
int64
```

To preserve dtypes while iterating over the rows, it is better to use *itertuples()* which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

it [generator] A generator that iterates over the rows of the frame.

itertuples : Iterate over DataFrame rows as namedtuples of the values. iteritems : Iterate over (column name, Series) pairs.

itertuples(*index=True*, *name='Pandas'*)

Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.

index [boolean, default True] If True, return the index as the first element of the tuple.

name [string, default "Pandas"] The name of the returned namedtuples or None to return regular tuples.

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. iteritems : Iterate over (column name, Series) pairs.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [0.1, 0.2]},
                      index=['a', 'b'])
>>> df
   col1  col2
a     1   0.1
b     2   0.2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='a', col1=1, col2=0.10000000000000001)
Pandas(Index='b', col1=2, col2=0.20000000000000001)
```

ix

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at Advanced Indexing.

join(*other*, *on=None*, *how='left'*, *lsuffix=''*, *rsuffix=''*, *sort=False*)

Join columns with other DataFrame either on index or on a key column. Efficiently Join multiple DataFrame objects by index at once by passing a list.

other [DataFrame, Series with name field set, or list of DataFrame] Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame

on [name, tuple/list of names, or array-like] Column or index level name(s) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

how [{'left', 'right', 'outer', 'inner'}, default: 'left'] How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use other frame's index
- outer: form union of calling frame's index (or column if on is specified) with other frame's index, and sort it lexicographically
- inner: form intersection of calling frame's index (or column if on is specified) with other frame's index, preserving the order of the calling's one

lsuffix [string] Suffix to use from left frame's overlapping columns

rsuffix [string] Suffix to use from right frame's overlapping columns

sort [boolean, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword)

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

Support for specifying index levels as the *on* parameter was added in version 0.23.0

```
>>> caller = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                        'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> caller
    A key
0  A0  K0
1  A1  K1
2  A2  K2
3  A3  K3
4  A4  K4
5  A5  K5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
    B key
0  B0  K0
1  B1  K1
2  B2  K2
```

Join DataFrames using their indexes.

```
>>> caller.join(other, lsuffix='_caller', rsuffix='_other')
```

```
>>>     A key_caller    B key_other
    0  A0         K0   B0        K0
    1  A1         K1   B1        K1
```

(continues on next page)

(continued from previous page)

```
    2  A2          K2   B2         K2
    3  A3          K3  NaN        NaN
    4  A4          K4  NaN        NaN
    5  A5          K5  NaN        NaN
```

If we want to join using the key columns, we need to set key to be the index in both caller and other. The joined DataFrame will have key as its index.

```
>>> caller.set_index('key').join(other.set_index('key'))
```

```
>>>      A    B
    key
    K0   A0   B0
    K1   A1   B1
    K2   A2   B2
    K3   A3  NaN
    K4   A4  NaN
    K5   A5  NaN
```

Another option to join using the key columns is to use the on parameter. DataFrame.join always uses other's index but we can use any column in the caller. This method preserves the original caller's index in the result.

```
>>> caller.join(other.set_index('key'), on='key')
```

```
>>>     A key    B
    0  A0  K0   B0
    1  A1  K1   B1
    2  A2  K2   B2
    3  A3  K3  NaN
    4  A4  K4  NaN
    5  A5  K5  NaN
```

DataFrame.merge : For column(s)-on-columns(s) operations

joined : DataFrame

keys()

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

kurt(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

kurtosis (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

last (*offset*)
Convenience method for subsetting final periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

subset : type of caller

first : Select initial periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

last_valid_index ()
Return index for last non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

le (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods le

loc
Access a group of rows and columns by label(s) or a boolean array.

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

> **Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at Selection by Label

DataFrame.at : Access a single value for a row/column label pair DataFrame.iloc : Access group of rows and columns by integer position(s) DataFrame.xs : Returns a cross-section (row(s) or column(s)) from the

Series/DataFrame.

Series.loc : Access group of values using labels

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=['cobra', 'viper', 'sidewinder'],
...       columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
```

(continues on next page)

(continued from previous page)

```
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

KeyError: when any items are not found

lookup (*row_labels*, *col_labels*)

Label-based "fancy indexing" function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

row_labels [sequence] The row labels to use for lookup

col_labels [sequence] The column labels to use for lookup

Akin to:

```
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

values [ndarray] The found values

lt(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods lt

mad(*axis=None*, *skipna=None*, *level=None*)
Return the mean absolute deviation of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mad : Series or DataFrame (if level specified)

mask(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

`DataFrame.where()`

max (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the maximum of the values in the object. If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

max : Series or DataFrame (if level specified)

mean (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mean : Series or DataFrame (if level specified)

median (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

median : Series or DataFrame (if level specified)

melt (*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *col_level=None*)

"Unpivots" a DataFrame from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

New in version 0.20.0.

frame : DataFrame id_vars : tuple, list, or ndarray, optional

Column(s) to use as identifier variables.

value_vars [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name [scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name [scalar, default 'value'] Name to use for the 'value' column.

col_level [int or string, optional] If columns are a MultiIndex then use this level to melt.

melt pivot_table DataFrame.pivot

```
>>> import pandas as pd
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> df.melt(id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

memory_usage(*index=True*, *deep=False*)

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

index [bool, default True] Specifies whether to include the memory usage of the DataFrame's index in returned Series. If `index=True` the memory usage of the index the first item in the output.

deep [bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

sizes [Series] A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

numpy.ndarray.nbytes [Total bytes consumed by the elements of an] ndarray.

Series.memory_usage : Bytes consumed by a Series. pandas.Categorical : Memory-efficient array for string values with

many repeated values.

DataFrame.info : Concise summary of a DataFrame.

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64  complex128 object  bool
0      1      1.0      (1+0j)      1  True
1      1      1.0      (1+0j)      1  True
2      1      1.0      (1+0j)      1  True
3      1      1.0      (1+0j)      1  True
4      1      1.0      (1+0j)      1  True
```

```
>>> df.memory_usage()
Index            80
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64         40000
float64       40000
complex128    80000
object        40000
```

(continues on next page)

(continued from previous page)

```
bool           5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index             80
int64          40000
float64        40000
complex128     80000
object        160000
bool            5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5168
```

merge(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *sort=False*, *suffixes=('_x', '_y')*, *copy=True*, *indicator=False*, *validate=None*)

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

right : DataFrame how : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys

on [label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [boolean, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

right_index [boolean, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index

sort [boolean, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword)

suffixes [2-length sequence (tuple, list, ...)] Suffix to apply to overlapping column names in the left and right side, respectively

copy [boolean, default True] If False, do not copy data unnecessarily

indicator [boolean or string, default False] If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in 'left' DataFrame, "right_only" for observations whose merge key only appears in 'right' DataFrame, and "both" if the observation's merge key is found in both.

validate [string, default None] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

New in version 0.21.0.

Support for specifying index levels as the *on*, *left_on*, and *right_on* parameters was added in version 0.23.0

```
>>> A              >>> B
    lkey value         rkey value
0   foo  1         0   foo  5
1   bar  2         1   bar  6
2   baz  3         2   qux  7
3   foo  4         3   bar  8
```

```
>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0  foo   1        foo   5
1  foo   4        foo   5
2  bar   2        bar   6
3  bar   2        bar   8
4  baz   3        NaN   NaN
5  NaN   NaN      qux   7
```

merged [DataFrame] The output type will the be same as 'left', if it is a subclass of DataFrame.

merge_ordered merge_asof DataFrame.join

merge_results(*others*)

Merges results of type *CleavageFragmentPredictionResult* and returns the merged result

Parameters **others** (list(*CleavageFragmentPredictionResult*) or *CleavageFragmentPredictionResult*) – A (list of) *CleavageFragmentPredictionResult* object(s)

Returns new merged *CleavageFragmentPredictionResult* object

Return type *CleavageFragmentPredictionResult*

min(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the minimum of the values in the object. If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min : Series or DataFrame (if level specified)

mod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmod

mode(*axis=0*, *numeric_only=False*)
Gets the mode(s) of each element along the axis selected. Adds a row for each mode per label, fills in gaps with nan.

Note that there could be multiple values returned for the selected axis (when more than one item share the maximum frequency), which is the reason why a dataframe is returned. If you want to impute missing values with the mode in a dataframe `df`, you can just do this: `df.fillna(df.mode().iloc[0])`

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index' : get mode of each column
- 1 or 'columns' : get mode of each row

numeric_only [boolean, default False] if True, only apply to numeric columns

modes : DataFrame (sorted)

```
>>> df = pd.DataFrame({'A': [1, 2, 1, 2, 1, 2, 3]})
>>> df.mode()
   A
0  1
1  2
```

mul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

multiply(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

ndim

Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

ndarray.ndim

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

ne(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods ne

nlargest (*n*, *columns*, *keep='first'*)

Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant.

n [int] Number of rows to return.

columns [label or list of labels] Column label(s) to order by.

keep [{'first', 'last'}, default 'first'] Where there are duplicate values:

- *first* : prioritize the first occurrence(s)
- *last* : prioritize the last occurrence(s)

DataFrame The first *n* rows ordered by the given columns in descending order.

DataFrame.nsmallest [Return the first *n* rows ordered by *columns* in] ascending order.

DataFrame.sort_values : Sort DataFrame by the values DataFrame.head : Return the first *n* rows without re-ordering.

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 10, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df
    a  b    c
0   1  a  1.0
1  10  b  2.0
2   8  d  NaN
3  10  c  3.0
4  -1  e  4.0
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column "a".

```
>>> df.nlargest(3, 'a')
    a  b    c
1  10  b  2.0
3  10  c  3.0
2   8  d  NaN
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'a', keep='last')
    a  b    c
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

To order by the largest values in column "a" and then "c", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['a', 'c'])
    a  b    c
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

Attempting to use `nlargest` on non-numeric dtypes will raise a `TypeError`:

```
>>> df.nlargest(3, 'b')
Traceback (most recent call last):
TypeError: Column 'b' has dtype object, cannot use method 'nlargest'
```

notna()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
```

(continues on next page)

(continued from previous page)

```
2    False
dtype: bool
```

notnull()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

nsmallest(*n*, *columns*, *keep='first'*)

Get the rows of a DataFrame sorted by the *n* smallest values of *columns*.

n [int] Number of items to retrieve

columns [list or str] Column name or names to order by

keep [{'first', 'last'}, default 'first'] Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

DataFrame

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 11, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nsmallest(3, 'a')
   a  b   c
4 -1  e   4
0  1  a   1
2  8  d NaN
```

nunique(*axis=0*, *dropna=True*)

Return Series with number of distinct observations over requested axis.

New in version 0.20.0.

axis : {0 or 'index', 1 or 'columns'}, default 0 dropna : boolean, default True

Don't include NaN in the counts.

nunique : Series

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 1, 1]})
>>> df.nunique()
A    3
B    1
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
```

pct_change(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs** Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

chg [Series or DataFrame] The same type as the calling object.

Series.diff : Compute the difference of two elements in a Series. DataFrame.diff : Compute the difference of two elements in a DataFrame. Series.shift : Shift the index by some number of periods. DataFrame.shift : Shift the index by some number of periods.

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...      'FR': [4.0405, 4.0963, 4.3149],
...      'GR': [1.7246, 1.7482, 1.8519],
...      'IT': [804.74, 810.01, 860.13]},
...      index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                  FR        GR        IT
1980-01-01       NaN       NaN       NaN
1980-02-01  0.013810  0.013684  0.006549
1980-03-01  0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...      '2016': [1769950, 30586265],
...      '2015': [1500923, 40912316],
...      '2014': [1371819, 41403351]},
...      index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pipe (*func*, **args*, ***kwargs*)

Apply func(self, *args, **kwargs)

func [function] function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the NDFrame.

args [iterable, optional] positional arguments passed into `func`.

kwargs [mapping, optional] a dictionary of keyword arguments passed into `func`.

object : the return type of `func`.

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pandas.DataFrame.apply pandas.DataFrame.applymap pandas.Series.map

pivot (*index=None*, *columns=None*, *values=None*)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the User Guide for more on reshaping.

index [string or object, optional] Column to use to make new frame's index. If None, uses existing index.

columns [string or object] Column to use to make new frame's columns.

values [string, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Changed in version 0.23.0: Also accept list of column names.

DataFrame Returns reshaped DataFrame.

ValueError: When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

DataFrame.pivot_table [generalization of pivot that can handle] duplicate values for one index/column pair.

DataFrame.unstack [pivot based on the index values instead of a] column.

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
...                    "baz": [1, 2, 3, 4]})
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

pivot_table(*values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*, *dropna=True*, *margins_name='All'*)

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame

values : column to aggregate, optional index : column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

columns [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

aggfunc [function, list of functions, dict, default numpy.mean] If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions

fill_value [scalar, default None] Value to replace missing values with

margins [boolean, default False] Add all row / columns (e.g. for subtotal / grand totals)

dropna [boolean, default True] Do not include columns whose entries are all NaN

margins_name [string, default 'All'] Name of the row / column that will contain the totals when margins is True.

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
>>> df
     A    B      C  D
0  foo  one  small  1
1  foo  one  large  2
```

(continues on next page)

(continued from previous page)

```
2  foo  one  large  2
3  foo  two  small  3
4  foo  two  small  3
5  bar  one  large  4
6  bar  one  small  5
7  bar  two  small  6
8  bar  two  large  7
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': [min, max, np.mean]})
>>> table
                  D   E
               mean max median min
A   C
bar large  5.500000  16   14.5  13
    small  5.500000  15   14.5  14
foo large  2.000000  10    9.5   9
    small  2.333333  12   11.0   8
```

table : DataFrame

DataFrame.pivot [pivot without aggregation that can handle] non-numeric data

plot

alias of `pandas.plotting._core.FramePlotMethods`

pop (*item*)

Return item and drop from frame. Raise KeyError if not found.

item [str] Column label to be popped

popped : Series

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
```

(continues on next page)

(continued from previous page)

```
...                    ('monkey', 'mammal', np.nan)],
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

pow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rpow

prod(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

product (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

quantile(*q=0.5*, *axis=0*, *numeric_only=True*, *interpolation='linear'*)

Return values at the given quantile over requested axis, a la numpy.percentile.

q [float or array-like, default 0.5 (50% quantile)] 0 <= q <= 1, the quantile(s) to compute

axis [{0, 1, 'index', 'columns'} (default 0)] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

numeric_only [boolean, default True] If False, the quantile of datetime and timedelta data will be computed as well

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.18.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

quantiles : Series or DataFrame

- If `q` is an array, a DataFrame will be returned where the index is `q`, the columns are the columns of self, and the values are the quantiles.
- If `q` is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                      columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
                       'B': [pd.Timestamp('2010'),
                             pd.Timestamp('2011')],
                       'C': [pd.Timedelta('1 days'),
                             pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
```

(continues on next page)

(continued from previous page)

```
C        1 days 12:00:00
Name: 0.5, dtype: object
```

pandas.core.window.Rolling.quantile

query(*expr*, *inplace=False*, ***kwargs*)

Query the columns of a frame with a boolean expression.

expr [string] The query string to evaluate. You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

inplace [bool] Whether the query should modify the data in place or return a modified copy

New in version 0.18.0.

kwargs [dict] See the documentation for `pandas.eval()` for complete details on the keyword arguments accepted by `DataFrame.query()`.

q : DataFrame

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level `pandas.eval()` function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the `query` documentation in indexing.

pandas.eval DataFrame.eval

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = pd.DataFrame(randn(10, 2), columns=list('ab'))
>>> df.query('a > b')
>>> df[df.a > df.b]  # same result as the previous expression
```

radd(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.add

rank (*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)
Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values

axis [{0 or 'index', 1 or 'columns'}, default 0] index to direct ranking

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

ranks : same type as caller

rdiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

reindex(***kwargs*)
Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like, optional] New labels / index to conform the axis specified by 'axis' to.

index, columns [array-like, optional (should be specified using keywords)] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis [int or str, optional] Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...       'http_status': [200,200,404,404,301],
...       'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...        index=index)
>>> df
            http_status  response_time
Firefox             200           0.04
Chrome              200           0.02
Safari              404           0.07
IE10                404           0.08
Konqueror           301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
```

(continues on next page)

(continued from previous page)

```
IE10                  404          0.08
Chrome                200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to backpropagate the last valid value to fill the NaN values, pass bfill as an argument to the method keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29     100
2009-12-30     100
2009-12-31     100
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

Please note that the NaN value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the NaN values present in the original dataframe, use the fillna() method.

See the user guide for more.

reindexed : DataFrame

reindex_axis (*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=nan*)

Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis : {0 or 'index', 1 or 'columns'} method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional

Method to use for filling holes in reindexed DataFrame:

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation abs(index[indexer] - target) <= tolerance.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

reindex, reindex_like

reindexed : DataFrame

reindex_like(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)
Return an object with matching indices to myself.

other : Object method : string or None copy : boolean, default True limit : int, default None

Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

reindexed : same as input

rename(***kwargs*)
Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the user guide for more.

mapper, index, columns [dict-like or function, optional] dict-like or functions transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

axis [int or str, optional] Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

copy [boolean, default True] Also copy underlying data

inplace [boolean, default False] Whether to return a new DataFrame. If True then value of copy is ignored.

level [int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

renamed : DataFrame

pandas.DataFrame.rename_axis

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
```

(continues on next page)

(continued from previous page)

```
1  2  5
2  3  6
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

rename_axis (*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter the name of the index or columns.

mapper [scalar, list-like, optional] Value to set as the axis name attribute.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis.

copy [boolean, default True] Also copy underlying data.

inplace [boolean, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

renamed [Series, DataFrame, or None] The same type as the caller or None if *inplace* is True.

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

pandas.Series.rename : Alter Series index labels or name pandas.DataFrame.rename : Alter DataFrame index labels or name pandas.Index.rename : Set new names on index

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s.rename_axis("foo")
foo
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")
     A  B
foo
0    1  4
1    2  5
2    3  6
```

```
>>> df.rename_axis("bar", axis="columns")
bar  A  B
0    1  4
1    2  5
2    3  6
```

reorder_levels(*order*, *axis=0*)

Rearrange index levels using input order. May not drop or duplicate levels

order [list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).

axis [int] Where to reorder levels.

type of caller (new object)

replace(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)

Replace values given in *to_replace* with *value*.

Values of the DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

to_replace [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be `None` to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:

- This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

value [scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace [boolean, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

regex [bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method [{'pad', 'ffill', 'bfill', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

DataFrame.fillna : Fill NA values DataFrame.where : Replace values based on boolean condition Series.str.replace : Simple string replacement.

DataFrame Object after replacement.

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Scalar 'to_replace' and 'value'

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like 'to_replace'

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
2    3
3    3
4    4
dtype: int64
```

dict-like 'to_replace'

```
>>> df.replace({0: 10, 1: 100})
     A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
     A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
     A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression 'to_replace'

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
      A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$':'new', 'foo':'xyz'})
      A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
      A    B
0   new  abc
1   new  new
2  bait  xyz
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
```

(continues on next page)

(continued from previous page)

```
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the pecularities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value=None` and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

resample(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

rule [string] the offset string or object representing target conversion

axis : int, optional, default 0 closed : {'right', 'left'}

Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label [{'right', 'left'}] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention [{'start', 'end', 's', 'e'}] For PeriodIndex only, controls whether to use the start or end of *rule*

kind: {'timestamp', 'period'}, optional Pass 'timestamp' to convert the resulting index to a `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`. By default the input representation is retained.

loffset [timedelta] Adjust the resampled time labels

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

on [string, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [string or int, optional] For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

Resampler object

See the user guide for more.

To learn more about the offset strings, please see this link.

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5] #select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like)+5
```

```
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))
>>> s
2012    1
```

(continues on next page)

(continued from previous page)

```
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using 'start' *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
                     a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                       columns=['a', 'b', 'c', 'd'],
                       index=pd.MultiIndex.from_product([time, [1, 2]])
                       )
>>> df2.resample('3T', level=0).sum()
                     a  b   c   d
2000-01-01 00:00:00  0  6  12  18
2000-01-01 00:03:00  0  4   8  12
```

groupby : Group by mapping, function, label, or list of labels.

reset_index(*level=None*, *drop=False*, *inplace=False*, *col_level=0*, *col_fill=''*)

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level_0' (if 'index' is already taken) will be used.

level [int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default

drop [boolean, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

col_level [int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill [object, default ''] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

resetted : DataFrame

```
>>> df = pd.DataFrame([('bird',    389.0),
...                    ('bird',     24.0),
...                    ('mammal',   80.5),
...                    ('mammal', np.nan)],
...                   index=['falcon', 'parrot', 'lion', 'monkey'],
...                   columns=('class', 'max_speed'))
>>> df
         class  max_speed
falcon    bird      389.0
parrot    bird       24.0
lion    mammal       80.5
monkey  mammal        NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
    class  max_speed
0    bird      389.0
1    bird       24.0
2  mammal       80.5
3  mammal        NaN
```

You can also use *reset_index* with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                    ('bird', 'parrot'),
...                                    ('mammal', 'lion'),
...                                    ('mammal', 'monkey')],
```

(continues on next page)

(continued from previous page)

```
...                                       names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                      ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
...                    ( 24.0, 'fly'),
...                    ( 80.5, 'run'),
...                    (np.nan, 'jump')],
...                   index=index,
...                   columns=columns)
>>> df
               speed species
                 max    type
class  name
bird   falcon  389.0     fly
       parrot   24.0     fly
mammal lion     80.5     run
       monkey    NaN    jump
```

If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
         class  speed species
                  max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
                speed species
         class    max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

When the index is inserted under another level, we can specify under which one with the parameter *col_fill*:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
              species  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

If we specify a nonexistent level for *col_fill*, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
                genus  speed species
                class    max    type
```

(continues on next page)

(continued from previous page)

```
name
falcon          bird  389.0     fly
parrot          bird   24.0     fly
lion          mammal   80.5     run
monkey        mammal    NaN    jump
```

rfloordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.floordiv

rmod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mod

rmul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mul

rolling(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculations.

New in version 0.18.0.

window [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. This is new in 0.19.0

min_periods [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

center [boolean, default False] Set the labels at the center of the window.

win_type [string, default None] Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

on [string, optional] For a DataFrame, column on which to calculate the rolling window, rather than the index

closed [string, default None] Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

axis : int or string, default 0

a Window or Rolling sub-classed for the particular operation

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
     B
0  NaN
1  1.0
```

(continues on next page)

(continued from previous page)

```
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                   index = [pd.Timestamp('20130101 09:00:00'),
...                            pd.Timestamp('20130101 09:00:02'),
...                            pd.Timestamp('20130101 09:00:03'),
...                            pd.Timestamp('20130101 09:00:05'),
...                            pd.Timestamp('20130101 09:00:06')])
```

```
>>> df
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```
>>> df.rolling('2s').sum()
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

To learn more about the offsets & frequency strings, please see this link.

The recognized win_types are:

- boxcar
- triang
- blackman
- hamming
- bartlett
- parzen
- bohman
- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general_gaussian (needs power, width)
- slepian (needs width).

If win_type=None all points are evenly weighted. To learn more about different window types see scipy.signal window functions.

expanding : Provides expanding transformations. ewm : Provides exponential weighted functions

round(*decimals=0*, **args*, ***kwargs*)

Round a DataFrame to a variable number of decimal places.

decimals [int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

```
>>> df = pd.DataFrame(np.random.random([3, 3]),
...     columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
>>> df
                A         B         C
first    0.028208  0.992815  0.173891
second   0.038683  0.645646  0.577595
third    0.877076  0.149370  0.491027
>>> df.round(2)
           A     B     C
first   0.03  0.99  0.17
second  0.04  0.65  0.58
third   0.88  0.15  0.49
>>> df.round({'A': 1, 'C': 2})
          A         B     C
first   0.0  0.992815  0.17
second  0.0  0.645646  0.58
third   0.9  0.149370  0.49
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
          A  B     C
first   0.0  1  0.17
```

(continues on next page)

(continued from previous page)

```
second  0.0  1  0.58
third   0.9  0  0.49
```

DataFrame object

numpy.around Series.round

rpow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.pow

rsub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
```

(continues on next page)

(continued from previous page)

```
...                   index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.sub

rtruediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*)

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

n [int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac [float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace [boolean, optional] Sample with or without replacement. Default = False.

weights [str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. inf and -inf values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

A new object of same type as caller.

Generate an example `Series` and `DataFrame`:

```
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0   -0.038497
1    1.820773
2   -0.972766
3   -1.598270
4   -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
          A         B         C         D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661
```

Next extract a random sample from both of these objects...

3 random elements from the `Series`:

```
>>> s.sample(n=3)
27   -0.994689
55   -1.049016
67   -0.224565
dtype: float64
```

And a random 10% of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.1, replace=True)
           A         B         C         D
35  1.981780  0.142106  1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640  0.217116
40  0.823173 -0.078816  1.009536  1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
6  -0.148339  0.832938  1.787600 -1.383767
```

You can use *random state* for reproducibility:

```
>>> df.sample(random_state=1)
A         B         C         D
37 -2.027662  0.103611  0.237496 -0.165867
43 -0.259323 -0.583426  1.516140 -0.479118
12 -1.686325 -0.579510  0.985195 -0.460286
8   1.167946  0.429082  1.215742 -1.636041
9   1.197475 -0.864188  1.554031 -1.505264
```

select (*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

crit [function] To be called on each index (label). Should return True or False

axis : int

selection : type of caller

select_dtypes(*include=None*, *exclude=None*)

Return a subset of the DataFrame's columns based on the column dtypes.

include, exclude [scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

ValueError

- If both of `include` and `exclude` are empty
- If `include` and `exclude` have overlapping elements
- If any kind of string dtype is passed in.

subset [DataFrame] The subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.

- To select all *numeric* types, use `np.number` or `'number'`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the numpy dtype hierarchy
- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`
- To select timedeltas, use `np.timedelta64`, `'timedelta'` or `'timedelta64'`
- To select Pandas categorical dtypes, use `'category'`
- To select Pandas datetimetz dtypes, use `'datetimetz'` (new in 0.20.0) or `'datetime64[ns, tz]'`

```
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
        a      b  c
0       1   True  1.0
1       2  False  2.0
2       1   True  1.0
3       2  False  2.0
4       1   True  1.0
5       2  False  2.0
```

```
>>> df.select_dtypes(include='bool')
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

```
>>> df.select_dtypes(include=['float64'])
   c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int'])
       b    c
0   True  1.0
1  False  2.0
2   True  1.0
3  False  2.0
4   True  1.0
5  False  2.0
```

sem(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

sem : Series or DataFrame (if level specified)

set_axis(*labels*, *axis=0*, *inplace=None*)
Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

labels [list-like, Index] The values for the new index.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

pandas.DataFrame.rename_axis : Alter the name of the index or columns.

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

set_index (*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

keys : column label or list of column labels / arrays drop : boolean, default True

Delete columns to be used as the new index

append [boolean, default False] Whether to append columns to existing index

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

verify_integrity [boolean, default False] Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale':[55, 40, 84, 31]})
   month  sale  year
0  1      55    2012
1  4      40    2014
2  7      84    2013
3  10     31    2014
```

Set the index to become the 'month' column:

```
>>> df.set_index('month')
       sale  year
month
1      55    2012
4      40    2014
7      84    2013
10     31    2014
```

Create a multi-index using columns 'year' and 'month':

```
>>> df.set_index(['year', 'month'])
            sale
year  month
2012  1     55
2014  4     40
2013  7     84
2014  10    31
```

Create a multi-index using a set of values and a column:

```
>>> df.set_index([[1, 2, 3, 4], 'year'])
         month  sale
   year
1  2012  1      55
2  2014  4      40
3  2013  7      84
4  2014  10     31
```

dataframe : DataFrame

set_value(*index*, *col*, *value*, *takeable=False*)

Put single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label value : scalar value takeable : interpret the index/col as indexers, default False

frame [DataFrame] If label pair is contained, will be reference to calling DataFrame, otherwise a new object

shape
Return a tuple representing the dimensionality of the DataFrame.

ndarray.shape

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

shift (*periods=1*, *freq=None*, *axis=0*)
Shift index by desired number of periods with an optional time freq

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, optional] Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

axis : {0 or 'index', 1 or 'columns'}

If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data.

shifted : DataFrame

size
Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

ndarray.size

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

skew (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

skew : Series or DataFrame (if level specified)

slice_shift (*periods=1*, *axis=0*)
Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

periods [int] Number of periods to move, can be positive or negative

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

shifted : same type as caller

sort_index (*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)
Sort object by labels (along an axis)

axis : index, columns to direct sorting level : int or level name or list of ints or list of level names

if not None, sort on values in specified index level(s)

ascending [boolean, default True] Sort ascending vs. descending

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining [bool, default True] if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

sorted_obj : DataFrame

sort_values (*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)
Sort by the values along either axis

by [str or list of str] Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels

Changed in version 0.23.0: Allow specifying index or column level names.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis to be sorted

ascending [bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end

sorted_obj : DataFrame

```
>>> df = pd.DataFrame({
...     'col1' : ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2' : [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
```

(continues on next page)

(continued from previous page)

```
... })
>>> df
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
3   NaN  8    4
4   D    7    2
5   C    4    3
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
    col1 col2 col3
1   A    1    1
0   A    2    0
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
    col1 col2 col3
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
3   NaN  8    4
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
    col1 col2 col3
3   NaN  8    4
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
```

sortlevel (*level=0*, *axis=0*, *ascending=True*, *inplace=False*, *sort_remaining=True*)

Sort multilevel index by chosen axis and primary level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order).

Deprecated since version 0.20.0: Use `DataFrame.sort_index()`

level : int axis : {0 or 'index', 1 or 'columns'}, default 0 ascending : boolean, default True inplace : boolean, default False

Sort the DataFrame without creating a new instance

sort_remaining [boolean, default True] Sort by the other levels too.

sorted : DataFrame

DataFrame.sort_index(level=...)

squeeze (*axis=None*)

Squeeze length 1 dimensions.

axis [None, integer or string axis name, optional] The axis to squeeze if 1-sized.

New in version 0.20.0.

scalar if 1-sized, else original object

stack (*level=-1*, *dropna=True*)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

The new index levels are sorted.

level [int, str, list, default -1] Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

dropna [bool, default True] Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of index and column values that are missing from the original dataframe. See Examples section.

DataFrame or Series Stacked dataframe or series.

DataFrame.unstack [Unstack prescribed level(s) from index axis] onto column axis.

DataFrame.pivot [Reshape dataframe from long format to wide] format.

DataFrame.pivot_table [Create a spreadsheet-style pivot table] as a DataFrame.

The function is named by analogy with a collection of books being re-organised from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of of each other (in the index of the dataframe).

Single level columns

```
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
     weight height
cat       0      1
dog       2      3
>>> df_single_level_cols.stack()
cat  weight    0
     height    1
dog  weight    2
     height    3
dtype: int64
```

Multi level columns: simple case

```
>>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol1)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
     weight
         kg    pounds
cat       1         2
dog       2         4
>>> df_multi_level_cols1.stack()
            weight
cat kg           1
    pounds       2
dog kg           2
    pounds       4
```

Missing values

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
    weight height
        kg      m
cat    1.0    2.0
dog    3.0    4.0
>>> df_multi_level_cols2.stack()
        height  weight
cat kg     NaN     1.0
    m      2.0     NaN
dog kg     NaN     3.0
    m      4.0     NaN
```

Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0)
             kg    m
cat height  NaN  2.0
    weight  1.0  NaN
dog height  NaN  4.0
    weight  3.0  NaN
>>> df_multi_level_cols2.stack([0, 1])
cat  height  m     2.0
     weight  kg    1.0
dog  height  m     4.0
     weight  kg    3.0
dtype: float64
```

Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
    weight height
        kg      m
cat    NaN    1.0
dog    2.0    3.0
>>> df_multi_level_cols3.stack(dropna=False)
        height  weight
cat kg     NaN     NaN
    m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
>>> df_multi_level_cols3.stack(dropna=True)
        height  weight
cat m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
```

std(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

std : Series or DataFrame (if level specified)

style
Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

pandas.io.formats.style.Styler

sub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

subtract(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the sum of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

sum : Series or DataFrame (if level specified)

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

swapaxes (*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately

y : same as input

swaplevel (*i=-2*, *j=-1*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

swapped : type of caller (new object)

Changed in version 0.18.1: The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

tail (*n=5*)

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

n [int, default 5] Number of rows to select.

type of caller The last *n* rows of the caller object.

pandas.DataFrame.head : The first *n* rows of the caller object.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

take (*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

indices [array-like] An array of ints indicating which positions to take.

axis [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

taken [type of caller] An array-like containing the elements taken from the object.

DataFrame.loc : Select a subset of a DataFrame by labels. DataFrame.iloc : Select a subset of a DataFrame by positions. numpy.take : Take elements from an array along an axis.

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                    columns=['name', 'class', 'max_speed'],
...                    index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

to_clipboard(*excel=True*, *sep=None*, ***kwargs*)

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

excel [bool, default True]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

sep [str, default `'\t'`] Field delimiter.

****kwargs** These parameters will be passed to DataFrame.to_csv.

DataFrame.to_csv [Write a DataFrame to a comma-separated values] (csv) file.

read_clipboard : Read text from clipboard and pass to read_table.

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

to_csv(*path_or_buf=None*, *sep=', '*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *encoding=None*, *compression=None*, *quoting=None*, *quotechar='"'*, *line_terminator='\n'*, *chunksize=None*, *tupleize_cols=None*, *date_format=None*, *doublequote=True*, *escapechar=None*, *decimal='.'*)

Write DataFrame to a comma-separated values (csv) file

path_or_buf [string or file handle, default None] File path or object, if None is provided the result is returned as a string.

sep [character, default ‘,’] Field delimiter for the output file.

na_rep [string, default ‘’] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R

mode [str] Python write mode, default ‘w’

encoding [string, optional] A string representing the encoding to use in the output file, defaults to ‘ascii’ on Python 2 and ‘utf-8’ on Python 3.

compression [string, optional] A string representing the compression to use in the output file. Allowed values are ‘gzip’, ‘bz2’, ‘zip’, ‘xz’. This input is only used when the first argument is a filename.

line_terminator [string, default `'\n'`] The newline character or character sequence to use in the output file

quoting [optional constant from csv module] defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric

quotechar [string (length 1), default ‘"’] character used to quote fields

doublequote [boolean, default True] Control quoting of *quotechar* inside a field

escapechar [string (length 1), default None] character used to escape *sep* and *quotechar* when appropriate

chunksize [int or None] rows to write at a time

tupleize_cols [boolean, default False] Deprecated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

Write MultiIndex columns as a list of tuples (if True) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if False).

date_format [string, default None] Format string for datetime objects

decimal: string, default '.' Character recognized as decimal separator. E.g. use ',' for European data

to_dense()
Return dense representation of NDFrame (as opposed to sparse)

to_dict(*orient='dict'*, *into=<type 'dict'>*)
Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

orient [str {'dict', 'list', 'series', 'split', 'records', 'index'}] Determines the type of the values of the dictionary.

- 'dict' (default) : dict like {column -> {index -> value}}
- 'list' : dict like {column -> [values]}
- 'series' : dict like {column -> Series(values)}
- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'records' : list like [{column -> value}, ... , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

into [class, default dict] The collections.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

New in version 0.21.0.

result : collections.Mapping like {column -> {index -> value}}

DataFrame.from_dict: create a DataFrame from a dictionary DataFrame.to_json: convert a DataFrame to JSON format

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   col1  col2
a     1   0.50
b     2   0.75
>>> df.to_dict()
{'col1': {'a': 1, 'b': 2}, 'col2': {'a': 0.5, 'b': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': a    1
         b    2
         Name: col1, dtype: int64,
 'col2': a    0.50
         b    0.75
         Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['a', 'b'], 'columns': ['col1', 'col2'],
 'data': [[1.0, 0.5], [2.0, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1.0, 'col2': 0.5}, {'col1': 2.0, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'a': {'col1': 1.0, 'col2': 0.5}, 'b': {'col1': 2.0, 'col2': 0.75}}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('a', 1), ('b', 2)])),
             ('col2', OrderedDict([('a', 0.5), ('b', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1.0, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2.0, 'col2': 0.75})]
```

to_excel(*excel_writer*, *sheet_name='Sheet1'*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *startrow=0*, *startcol=0*, *engine=None*, *merge_cells=True*, *encoding=None*, *inf_rep='inf'*, *verbose=True*, *freeze_panes=None*)

Write DataFrame to an excel sheet

excel_writer [string or ExcelWriter object] File path or existing ExcelWriter

sheet_name [string, default 'Sheet1'] Name of sheet which will contain DataFrame

na_rep [string, default ''] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow : upper left cell row to dump data frame

startcol : upper left cell column to dump data frame

engine [string, default None] write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

merge_cells [boolean, default True] Write MultiIndex and Hierarchical Rows as merged cells.

encoding: string, default None encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep [string, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel)

freeze_panes [tuple of integer (length 2), default None] Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

For compatibility with to_csv, to_excel serializes lists and dicts to strings before writing.

to_feather(*fname*)
write out the binary feather-format for DataFrames

New in version 0.20.0.

fname [str] string file path

to_gbq(*destination_table*, *project_id*, *chunksize=None*, *verbose=None*, *reauth=False*, *if_exists='fail'*, *private_key=None*, *auth_local_webserver=False*, *table_schema=None*)
Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

Authentication to the Google BigQuery service is via OAuth 2.0.

- If `private_key` is provided, the library loads the JSON service account credentials and uses those to authenticate.
- If no `private_key` is provided, the library tries application default credentials.
- If application default credentials are not found or cannot be used with BigQuery, the library authenticates with user account credentials. In this case, you will be asked to grant permissions for product name 'pandas GBQ'.

destination_table [str] Name of table to be written, in the form 'dataset.tablename'.

project_id [str] Google BigQuery Account project ID.

chunksize [int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

reauth [bool, default False] Force Google BigQuery to reauthenticate the user. This is useful if multiple accounts are used.

if_exists [str, default 'fail'] Behavior when the destination table exists. Value can be one of:

'fail' If table exists, do nothing.

'replace' If table exists, drop it, recreate it, and insert data.

'append' If table exists, insert data. Create if does not exist.

private_key [str, optional] Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

auth_local_webserver [bool, default False] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

table_schema [list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'},...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

New in version 0.3.1 of pandas-gbq.

verbose [boolean, deprecated] *Deprecated in Pandas-GBQ 0.4.0.* Use the logging module to adjust verbosity instead.

pandas_gbq.to_gbq : This function in the pandas-gbq library. pandas.read_gbq : Read a DataFrame from Google BigQuery.

to_hdf (*path_or_buf*, *key*, ***kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the user guide.

path_or_buf [str or pandas.HDFStore] File path or HDFStore object.

key [str] Identifier for the group in the store.

mode [{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

format [{'fixed', 'table'}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

append [bool, default False] For Table formats, append the input data to the existing.

data_columns [list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See io.hdf5-query-data-columns. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

errors [str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

DataFrame.read_hdf : Read from HDF file. DataFrame.to_parquet : Write a DataFrame to the binary parquet format. DataFrame.to_sql : Write to a sql table. DataFrame.to_feather : Write out feather-format for DataFrames. DataFrame.to_csv : Write out to a csv file.

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

to_html (*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *bold_rows=True*, *classes=None*, *escape=True*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*, *notebook=False*, *decimal='.'*, *border=None*, *table_id=None*)

Render a DataFrame as an HTML table.

to_html-specific options:

bold_rows [boolean, default True] Make the row labels bold in the output

classes [str or list or tuple, default None] CSS class(es) to apply to the resulting html table

escape [boolean, default True] Convert the characters <, >, and & to HTML-safe sequences.

max_rows [int, optional] Maximum number of rows to show before truncating. If None, show all.

max_cols [int, optional] Maximum number of columns to show before truncating. If None, show all.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe

New in version 0.18.0.

border [int] A `border=border` attribute is included in the opening <*table*> tag. Default `pd.options.html.border`.

New in version 0.19.0.

table_id [str, optional] A css id is included in the opening <*table*> tag if specified.

New in version 0.23.0.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] whether to print column labels, default True

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_json(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression=None*, *index=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

path_or_buf [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'

  - allowed values are: {‘split’,’records’,’index’}
- DataFrame
  - default is ‘columns’
  - allowed values are: {‘split’,’records’,’index’,’columns’,’values’}
- The format of the JSON string
  - ‘split’ : dict like {‘index’ -> [index], ‘columns’ -> [columns], ‘data’ -> [values]}
  - ‘records’ : list like [{column -> value}, … , {column -> value}]
  - ‘index’ : dict like {index -> {column -> value}}
  - ‘columns’ : dict like {column -> {index -> value}}
  - ‘values’ : just the values array
  - ‘table’ : dict like {‘schema’: {schema}, ‘data’: {data}} describing the data, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

date_format [{None, ‘epoch’, ‘iso’}] Type of date conversion. ‘epoch’ = epoch milliseconds, ‘iso’ = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is ‘iso’. For all other orients, the default is ‘epoch’.

double_precision [int, default 10] The number of decimal places to use when encoding floating point values.

force_ascii [boolean, default True] Force encoded string to be ASCII.

date_unit [string, default ‘ms’ (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’, ‘ns’ for second, millisecond, microsecond, and nanosecond respectively.

default_handler [callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines [boolean, default False] If ‘orient’ is ‘records’ write out line delimited json format. Will throw ValueError if incorrect ‘orient’ since others are not list like.

New in version 0.19.0.

compression [{None, ‘gzip’, ‘bz2’, ‘zip’, ‘xz’}] A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

index [boolean, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is ‘split’ or ‘table’.

New in version 0.23.0.

pandas.read_json

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]}'
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

to_latex(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=False*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires \usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

to_latex-specific options:

bold_rows [boolean, default False] Make the row labels bold in the output

column_format [str, default None] The columns format as specified in LaTeX table format e.g 'rcl' for 3 columns

longtable [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a \usepackage{longtable} to your LaTeX preamble.

escape [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

encoding [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

multicolumn [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module.

New in version 0.20.0.

multirow [boolean, default False] Use multirow to enhance MultiIndex rows. Requires adding a \usepackage{multirow} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

New in version 0.20.0.

to_msgpack (*path_or_buf=None*, *encoding='utf-8'*, ***kwargs*)
msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

path [string File path, buffer-like, or None] if None, return generated string

append [boolean whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

to_panel ()
Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Deprecated since version 0.20.0.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

panel : Panel

to_parquet (*fname*, *engine='auto'*, *compression='snappy'*, ***kwargs*)
Write a DataFrame to the binary parquet format.

New in version 0.21.0.

This function writes the dataframe as a parquet file. You can choose different parquet backends, and have the option of compression. See the user guide for more details.

fname [str] String file path.

engine [{'auto', 'pyarrow', 'fastparquet'}, default 'auto'] Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

compression [{'snappy', 'gzip', 'brotli', None}, default 'snappy'] Name of the compression to use. Use `None` for no compression.

****kwargs** Additional arguments passed to the parquet library. See pandas io for more details.

read_parquet : Read a parquet file. DataFrame.to_csv : Write a csv file. DataFrame.to_sql : Write to a sql table. DataFrame.to_hdf : Write to hdf.

This function requires either the fastparquet or pyarrow library.

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gzip', compression='gzip')
>>> pd.read_parquet('df.parquet.gzip')
   col1  col2
```

(continues on next page)

(continued from previous page)

```
0      1      3
1      2      4
```

to_period (*freq=None*, *axis=0*, *copy=True*)

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

freq : string, default axis : {0 or 'index', 1 or 'columns'}, default 0

The axis to convert (the index by default)

copy [boolean, default True] If False then underlying input data is not copied

ts : TimeSeries with PeriodIndex

to_pickle (*path*, *compression='infer'*, *protocol=2*)

Pickle (serialize) object to file.

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

protocol [int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1]_ paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

read_pickle : Load pickled pandas object (or any object) from file. DataFrame.to_hdf : Write DataFrame to an HDF5 file. DataFrame.to_sql : Write DataFrame to a SQL database. DataFrame.to_parquet : Write a DataFrame to the binary parquet format.

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

to_records(*index=True*, *convert_datetime64=None*)

Convert DataFrame to a NumPy record array.

Index will be put in the 'index' field of the record array if requested.

index [boolean, default True] Include index in resulting record array, stored in 'index' field.

convert_datetime64 [boolean, default None] Deprecated since version 0.23.0.

Whether to convert the index to datetime.datetime if it is a DatetimeIndex.

y : numpy.recarray

DataFrame.from_records: convert structured or record ndarray to DataFrame.

numpy.recarray: ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   A     B
a  1  0.50
b  2  0.75
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

By default, timestamps are converted to *datetime.datetime*:

```
>>> df.index = pd.date_range('2018-01-01 09:00', periods=2, freq='min')
>>> df
                     A     B
2018-01-01 09:00:00  1  0.50
2018-01-01 09:01:00  2  0.75
>>> df.to_records()
rec.array([(datetime.datetime(2018, 1, 1, 9, 0), 1, 0.5 ),
           (datetime.datetime(2018, 1, 1, 9, 1), 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The timestamp conversion can be disabled so NumPy's datetime64 data type is used instead:

```
>>> df.to_records(convert_datetime64=False)
rec.array([('2018-01-01T09:00:00.000000000', 1, 0.5 ),
           ('2018-01-01T09:01:00.000000000', 2, 0.75)],
          dtype=[('index', '<M8[ns]'), ('A', '<i8'), ('B', '<f8')])
```

to_sparse(*fill_value=None*, *kind='block'*)

Convert to SparseDataFrame

fill_value : float, default NaN kind : {'block', 'integer'}

y : SparseDataFrame

to_sql(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1]_ are supported. Tables can be newly created, appended to, or overwritten.

name [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists [{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index [boolean, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize [int, optional] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

ValueError When the table already exists and *if_exists* is 'fail' (the default).

pandas.read_sql : read a DataFrame from a table

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
```

(continues on next page)

(continued from previous page)

```
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

to_stata(*fname*, *convert_dates=None*, *write_index=True*, *encoding='latin-1'*, *byteorder=None*, *time_stamp=None*, *data_label=None*, *variable_labels=None*, *version=114*, *convert_strl=None*)

Export Stata binary dta files.

fname [path (string), buffer or path object] string, path object (pathlib.Path or py._path.local.LocalPath) or object implementing a binary write() functions. If using a buffer then the buffer will not be automatically closed after the file data has been written.

convert_dates [dict] Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to 'tc'. Raises NotImplementedError if a datetime column has timezone information.

write_index [bool] Write the index to Stata dataset.

encoding [str] Default is latin-1. Unicode is not supported.

byteorder [str] Can be ">", "<", "little", or "big". default is *sys.byteorder*.

time_stamp [datetime] A datetime to use as file creation date. Default is the current time.

data_label [str] A label for the data set. Must be 80 characters or smaller.

variable_labels [dict] Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

New in version 0.19.0.

version [{114, 117}] Version to use in the output dta file. Version 114 can be used read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 114 limits string variables to 244 characters

or fewer while 117 allows strings with lengths up to 2,000,000 characters.

New in version 0.23.0.

convert_strl [list, optional] List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

New in version 0.23.0.

NotImplementedError

- If datetimes contain timezone information
- Column dtype is not representable in Stata

ValueError

- Columns listed in convert_dates are neither datetime64[ns] or datetime.datetime
- Column listed in convert_dates is not in DataFrame
- Categorical label contains more than 32,000 characters

New in version 0.19.0.

pandas.read_stata : Import Stata data files pandas.io.stata.StataWriter : low-level writer for Stata data files pandas.io.stata.StataWriter117 : low-level writer for version 117 files

```
>>> data.to_stata('./data_file.dta')
```

Or with dates

```
>>> data.to_stata('./date_data_file.dta', {2 : 'tw'})
```

Alternatively you can create an instance of the StataWriter class

```
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

With dates:

```
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
```

to_string(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *line_width=None*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*)

Render a DataFrame to a console-friendly tabular output.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_timestamp (*freq=None*, *how='start'*, *axis=0*, *copy=True*)
Cast to DatetimeIndex of timestamps, at *beginning* of period

freq [string, default frequency of PeriodIndex] Desired frequency

how [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default)

copy [boolean, default True] If false then underlying input data is not copied

df : DataFrame with DatetimeIndex

to_xarray()
Return an xarray object from the pandas object.

a DataArray for a Series a Dataset for a DataFrame a DataArray for higher dims

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
```

(continues on next page)

(continued from previous page)

```
   A    B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index    (index) int64 0 1 2
Data variables:
    A        (index) int64 1 1 2
    B        (index) object 'foo' 'bar' 'foo'
    C        (index) float64 4.0 5.0 6.0
```

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)}
                      ).set_index(['B','A'])
>>> df
         C
B   A
foo 1  4.0
bar 1  5.0
foo 2  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B        (B) object 'bar' 'foo'
  * A        (A) int64 1 2
Data variables:
    C        (B, A) float64 5.0 nan 4.0 6.0
```

```
>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                 items=list('ABCD'),
                 major_axis=pd.date_range('20130101', periods=3),
                 minor_axis=['first', 'second'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second
```

```
>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],
       [[ 6,  7],
        [ 8,  9],
        [10, 11]],
       [[12, 13],
```

(continues on next page)

(continued from previous page)

```
        [14, 15],
        [16, 17]],
       [[18, 19],
        [20, 21],
        [22, 23]]])
Coordinates:
  * items       (items) object 'A' 'B' 'C' 'D'
  * major_axis  (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03
↪# noqa
  * minor_axis  (minor_axis) object 'first' 'second'
```

See the xarray docs

transform(*func*, **args*, ***kwargs*)

Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values

New in version 0.20.0.

func [callable, string, dictionary, or list of string/callables] To apply to column

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

transformed : NDFrame

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
...                   index=pd.date_range('1/1/2000', periods=10))
df.iloc[3:7] = np.nan
```

```
>>> df.transform(lambda x: (x - x.mean()) / x.std())
                   A         B         C
2000-01-01  0.579457  1.236184  0.123424
2000-01-02  0.370357 -0.605875 -1.231325
2000-01-03  1.455756 -0.277446  0.288967
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -0.498658  1.274522  1.642524
2000-01-09 -0.540524 -1.012676 -0.828968
2000-01-10 -1.366388 -0.614710  0.005378
```

pandas.NDFrame.aggregate pandas.NDFrame.apply

transpose(**args*, ***kwargs*)

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
                0     1
name        Alice   Bob
score         9.5     8
employed    False  True
kids            0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

truediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

before [date, string, int] Truncate all rows before this index value.

after [date, string, int] Truncate all rows after this index value.

axis [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

copy [boolean, default is True,] Return a copy of the truncated section.

type of caller The truncated Series or DataFrame.

DataFrame.loc : Select a subset of a DataFrame by label. DataFrame.iloc : Select a subset of a DataFrame by position.

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
```

(continues on next page)

(continued from previous page)

```
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

tshift (*periods=1*, *freq=None*, *axis=0*)
Shift the time index, using the index's frequency if available.

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

shifted : NDFrame

tz_convert (*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

tz : string or pytz.timezone object axis : the axis to convert level : int, str, default None

If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

TypeError If the axis is tz-naive.

tz_localize (*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*)
Localize tz-naive TimeSeries to target time zone.

tz : string or pytz.timezone object axis : the axis to localize level : int, str, default None

If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise']

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times

- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

TypeError If the TimeSeries is tz-aware and tz is not None.

unstack (*level=-1*, *fill_value=None*)

Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels. If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex). The level involved will automatically get sorted.

level [int, string, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name

fill_value [replace NaN with this value if the unstack produces] missing values

New in version 0.18.0.

DataFrame.pivot : Pivot a table based on column values. DataFrame.stack : Pivot a level of the column labels (inverse operation

from *unstack*).

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                    ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a   1.0
     b   2.0
two  a   3.0
     b   4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a   b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
   one  two
a  1.0   3.0
b  2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a  1.0
     b  2.0
two  a  3.0
     b  4.0
dtype: float64
```

unstacked : DataFrame or Series

update (*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

other [DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

join [{'left'}, default 'left'] Only left join is implemented, keeping the index and columns of the original object.

overwrite [bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.
- False: only update values that are NA in the original DataFrame.

filter_func [callable(1d-array) -> boolean 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

raise_conflict [bool, default False] If True, will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

ValueError When *raise_conflict* is True and there's overlapping non-NA data.

dict.update : Similar method for dictionaries. DataFrame.merge : For column(s)-on-columns(s) operations.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
1  b  e
2  c  f
```

For Series, it's name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
```

(continues on next page)

(continued from previous page)

```
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A      B
0  1    4.0
1  2  500.0
2  3    6.0
```

values

Return a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

numpy.ndarray The values of the DataFrame.

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age':    [ 3,  29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3      94      31
1   29     170     115
>>> df.dtypes
age       int64
height    int64
weight    int64
dtype: object
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('monkey', np.nan, None)],
...                    columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name          object
max_speed    float64
rank          object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
```

(continues on next page)

(continued from previous page)

```
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

pandas.DataFrame.index : Retrievie the index labels pandas.DataFrame.columns : Retrieving the column names

var(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

var : Series or DataFrame (if level specified)

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
```

(continues on next page)

(continued from previous page)

```
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

DataFrame.mask()

xs(*key*, *axis=0*, *level=None*, *drop_level=True*)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on the rows (axis=0).

key [object] Some label contained in the index, or partially in a MultiIndex

axis [int, default 0] Axis to retrieve cross-section on

level [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level [boolean, default True] If False, returns object with same levels as self.

```
>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
A    4
B    5
C    2
Name: a
>>> df.xs('C', axis=1)
a    2
b    9
c    3
Name: C
```

```
>>> df
                     A  B  C  D
first second third
bar   one    1       4  1  8  9
      two    1       7  5  5  0
baz   one    1       6  6  8  0
      three  2       5  3  5  3
>>> df.xs(('baz', 'three'))
       A  B  C  D
third
2      5  3  5  3
>>> df.xs('one', level=1)
             A  B  C  D
first third
bar   1      4  1  8  9
baz   1      6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
        A  B  C  D
```

(continues on next page)

(continued from previous page)

```
second
three   5  3  5  3
```

xs : Series or DataFrame

xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of xs functionality, see MultiIndex Slicers

class Fred2.Core.Result.**CleavageSitePredictionResult**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

Bases: *Fred2.Core.Result.AResult*

A *CleavageSitePredictionResult* object is a `pandas.DataFrame` with multi-indexing, where column Ids are the prediction scores fo the different prediction methods, as well as the amino acid a a specific position, row ID the *Protein* ID and the position of the sequence (starting at 0).

CleavageSitePredictionResult:

ID	Pos	Seq	Method_name
protein_ID	0	S	0.56
	1	Y	15
	2	F	0.36
	3	P	10

T

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
```

(continues on next page)

(continued from previous page)

```
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
              0     1
name      Alice   Bob
score       9.5     8
employed  False  True
kids          0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

abs()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

abs Series/DataFrame containing the absolute value of each element.

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
     a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
     a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

numpy.absolute : calculate the absolute value element-wise.

add(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.radd

add_prefix(*prefix*)

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

prefix [str] The string to add before each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_suffix: Suffix row labels with string *suffix*. DataFrame.add_suffix: Suffix column labels with string *suffix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0       1       3
1       2       4
2       3       5
3       4       6
```

add_suffix(*suffix*)

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

suffix [str] The string to add after each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_prefix: Prefix row labels with string *prefix*. DataFrame.add_prefix: Prefix column labels with string *prefix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

agg (*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

aggregate (*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

align(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)

Align two objects on their axes with the specified join method for each axis Index

other : DataFrame or Series join : {'outer', 'inner', 'left', 'right'}, default 'outer' axis : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None)

level [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level

copy [boolean, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

method : str, default None limit : int, default None fill_axis : {0 or 'index', 1 or 'columns'}, default 0

Filling axis, method and limit

broadcast_axis [{0 or 'index', 1 or 'columns'}, default None] Broadcast values along this axis, if aligning two objects of different dimensions

(left, right) [(DataFrame, type of other)] Aligned objects

all (*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether all elements are True, potentially over an axis.

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

all : Series or DataFrame (if level specified)

pandas.Series.all : Return True if all elements are True pandas.DataFrame.any : Return True if one (or more) elements are True

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

any(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

any : Series or DataFrame (if level specified)

pandas.DataFrame.all : Return whether all elements are True.

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([True, False]).any()
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
```

(continues on next page)

(continued from previous page)

```
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

append(*other*, *ignore_index=False*, *verify_integrity=False*, *sort=None*)

Append rows of *other* to the end of this frame, returning a new object. Columns not in this frame are added as new columns.

other [DataFrame or Series/dict-like object, or list of these] The data to append.

ignore_index [boolean, default False] If True, do not use the index labels.

verify_integrity [boolean, default False] If True, raise ValueError on creating index with duplicates.

sort [boolean, default None] Sort columns if the columns of *self* and *other* are not aligned. The default sorting is deprecated and will change to not-sorting in a future version of pandas. Explicitly pass `sort=True` to silence the warning and sort. Explicitly pass `sort=False` to silence the warning and not sort.

New in version 0.23.0.

appended : DataFrame

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

pandas.concat [General function to concatenate DataFrame, Series] or Panel objects

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With *ignore_index* set to True:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
...     df = df.append({'A': i}, ignore_index=True)
>>> df
   A
0  0
1  1
2  2
3  3
4  4
```

More efficient:

```
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
...           ignore_index=True)
   A
0  0
1  1
2  2
3  3
4  4
```

apply (*func*, *axis=0*, *broadcast=None*, *raw=False*, *reduce=None*, *result_type=None*, *args=()*, ***kwds*)
Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

func [function] Function to apply to each column or row.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

broadcast [bool, optional] Only relevant for aggregation functions:

- `False` or `None` : returns a Series whose length is the length of the index or the number of columns (based on the *axis* parameter)
- `True` : results will be broadcast to the original shape of the frame, the original index and columns will be retained.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by result_type='broadcast'.

raw [bool, default False]

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

reduce [bool or None, default None] Try to apply reduction procedures. If the DataFrame is empty, *apply* will use *reduce* to determine whether the result should be a Series or a DataFrame. If `reduce=None` (the default), *apply*'s return value will be guessed by calling *func* on an empty Series (note: while guessing, exceptions raised by *func* will be ignored). If `reduce=True` a Series will always be returned, and if `reduce=False` a DataFrame will always be returned.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by `result_type='reduce'`.

result_type [{'expand', 'reduce', 'broadcast', None}, default None] These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.
- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

New in version 0.23.0.

args [tuple] Positional arguments to pass to *func* in addition to the array/series.

****kwds** Additional keyword arguments to pass as keywords arguments to *func*.

In the current implementation apply calls *func* twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if *func* has side-effects, as they will take effect twice for the first column/row.

DataFrame.applymap: For elementwise operations DataFrame.aggregate: only perform aggregating type operations DataFrame.transform: only perform transformating type operations

```
>>> df = pd.DataFrame([[4, 9],] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
```

(continues on next page)

(continued from previous page)

```
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Retuning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing result_type='expand' will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
```

(continues on next page)

(continued from previous page)

```
1  1  2
2  1  2
```

applied : Series or DataFrame

applymap (*func*)

Apply a function to a Dataframe elementwise.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

func [callable] Python function, returns a single value from a single value.

DataFrame Transformed DataFrame.

DataFrame.apply : Apply a function along input axis of DataFrame

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.applymap(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.applymap(lambda x: x**2)
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

But it's better to avoid applymap in that case.

```
>>> df ** 2
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

as_blocks (*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

copy : boolean, default True

values : a dict of dtype -> Constructor Types

as_matrix (*columns=None*)

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

columns: list, optional, default:None If None, return all columns, otherwise, returns specified columns.

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a flot64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

pandas.DataFrame.values

asfreq(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)
Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

freq : DateOffset object, or string method : {'backfill'/'bfill', 'pad'/'ffill'}, default None

Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill

how [{'start', 'end'}, default end] For PeriodIndex only, see PeriodIndex.asfreq

normalize [bool, default False] Whether to reset output index to midnight

fill_value: scalar, optional Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

converted : type of caller

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:03:00    3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                     s
2000-01-01 00:00:00    0.0
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    2.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    3.0
2000-01-01 00:03:00    3.0
```

reindex

To learn more about the frequency strings, please see this link.

asof (*where*, *subset=None*)

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

where : date or array of dates subset : string or list of strings, default None

if not None use these columns for NaN propagation

Dates are assumed to be sorted Raises if this is not the case

where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

where is Index: same shape object as input

merge_asof

assign (***kwargs*)

Assign new columns to a DataFrame, returning a new object (a copy) with the new columns added to the original ones. Existing columns that are re-assigned will be overwritten.

kwargs [keyword, value pairs] keywords are the column names. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

df [DataFrame] A new DataFrame with the new columns in addition to all the existing columns.

Assigning multiple columns within the same `assign` is possible. For Python 3.6 and above, later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order. For Python 3.5 and below, the order of keyword arguments is not specified, you cannot refer to newly created or modified columns. All items are computed first, and then assigned in alphabetical order.

Changed in version 0.23.0: Keyword argument order is maintained for Python 3.6 and later.

```
>>> df = pd.DataFrame({'A': range(1, 11), 'B': np.random.randn(10)})
```

Where the value is a callable, evaluated on *df*:

```
>>> df.assign(ln_A = lambda x: np.log(x.A))
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the value already exists and is inserted:

```
>>> newcol = np.log(df['A'])
>>> df.assign(ln_A=newcol)
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the keyword arguments depend on each other

```
>>> df = pd.DataFrame({'A': [1, 2, 3]})
```

```
>>> df.assign(B=df.A, C=lambda x:x['A']+ x['B'])
    A  B  C
 0  1  1  2
 1  2  2  4
 2  3  3  6
```

astype(***kwargs*)

Cast a pandas object to a specified dtype `dtype`.

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy [bool, default True.] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors [{'raise', 'ignore'}, default 'raise'.] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

raise_on_error [raise on invalid input] Deprecated since version 0.20.0: Use `errors` instead

kwargs : keyword arguments to pass on to the constructor

casted : type of caller

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> ser.astype('category', ordered=True, categories=[2, 1])
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to a numeric type. numpy.ndarray.astype : Cast a numpy array to a specified type.

at

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.iat [Access a single value for a row/column pair by integer] position

DataFrame.loc : Access a group of rows and columns by label(s) Series.at : Access a single value using a label

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

KeyError When label does not exist in DataFrame

at_time (*time*, *asof=False*)

Select values at particular time of day (e.g. 9:30AM).

TypeError If the index is not a `DatetimeIndex`

time : datetime.time or string

values_at_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

between_time : Select values between particular times of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_at_time : Get just the index locations for

values at particular time of the day

axes

Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['col1', 'col2'],
dtype='object')]
```

between_time (*start_time*, *end_time*, *include_start=True*, *include_end=True*)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

TypeError If the index is not a `DatetimeIndex`

start_time : datetime.time or string end_time : datetime.time or string include_start : boolean, default True include_end : boolean, default True

values_between_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

at_time : Select values at a particular time of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_between_time : Get just the index locations for

values between particular times of the day

bfill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='bfill')`

blocks
Internal property, property synonym for as_blocks()

Deprecated since version 0.21.0.

bool()
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

boxplot(*column=None*, *by=None*, *ax=None*, *fontsize=None*, *rot=0*, *grid=True*, *figsize=None*, *layout=None*, *return_type=None*, ***kwds*)
Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to *1.5 * IQR (IQR = Q3 - Q1)* from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia's entry for boxplot.

column [str or list of str, optional] Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.

by [str or array-like, optional] Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in *by*.

ax [object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.

fontsize [float or str] Tick label font size in points or as a string (e.g., *large*).

rot [int or float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate sytem.

grid [boolean, default True] Setting this to True will show the grid.

figsize [A tuple (width, height) in inches] The size of the figure to create in matplotlib.

layout [tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 columns and 5 rows, starting from the top-left.

return_type [{'axes', 'dict', 'both'} or None, default 'axes'] The kind of object to return. The default is `axes`.

- 'axes' returns the matplotlib axes the boxplot is drawn on.
- 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot.
- 'both' returns a namedtuple with the axes and dict.
- when grouping with `by`, a Series mapping columns to `return_type` is returned.

  If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

result :

The return type depends on the *return_type* parameter:

- 'axes' : object of class matplotlib.axes.Axes

- 'dict' : dict of matplotlib.lines.Line2D objects
- 'both' : a nametuple with strucure (ax, lines)

For data grouped with `by`:

- `Series`
- `array` (for `return_type = None`)

Series.plot.hist: Make a histogram. matplotlib.pyplot.boxplot : Matplotlib equivalent plot.

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`. For instance:

A list of strings (i.e. `['X', 'Y']`) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

The layout of boxplot can be adjusted giving a tuple to `layout`:

Additional formatting can be done to the boxplot, like suppressing the grid (`grid=False`), rotating the labels in the x-axis (i.e. `rot=45`) or changing the fontsize (i.e. `fontsize=15`):

The parameter `return_type` can be used to select the type of element returned by *boxplot*. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```
>>> boxplot = df.boxplot(column=['Col1','Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

When grouping with `by`, a Series mapping columns to `return_type` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned:

```
>>> boxplot =  df.boxplot(column=['Col1', 'Col2'], by='X',
...                       return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

`clip`(*lower=None*, *upper=None*, *axis=None*, *inplace=False*, **args*, ***kwargs*)
Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

lower [float or array_like, default None] Minimum threshold value. All values below this threshold will be set to it.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

clip_lower : Clip values below specified threshold(s). clip_upper : Clip values above specified threshold(s).

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

clip_lower(*threshold*, *axis=None*, *inplace=False*)

Return copy of the input with values below a threshold truncated.

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Series.clip [Return copy of input with values below and above] thresholds truncated.

Series.clip_upper [Return copy of input with values above] threshold truncated.

clipped : same type as input

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

clip_upper (*threshold*, *axis=None*, *inplace=False*)

Return copy of input with values above given value(s) truncated.

threshold : float or array_like axis : int or string axis name, optional

Align object with threshold along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data

New in version 0.21.0.

clip

clipped : same type as input

columns

The column labels of the DataFrame.

combine (*other*, *func*, *fill_value=None*, *overwrite=True*)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

other : DataFrame func : function

Function that takes two series as inputs and return a Series or a scalar

fill_value : scalar value overwrite : boolean, default True

If True then overwrite values for common keys in the calling frame

result : DataFrame

```
>>> df1 = DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, lambda s1, s2: s1 if s1.sum() < s2.sum() else s2)
   A  B
0  0  3
1  0  3
```

DataFrame.combine_first [Combine two DataFrame objects and default to] non-null values in frame calling the method

combine_first (*other*)

Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

other : DataFrame

combined : DataFrame

df1's values prioritized, use values from df2 to fill holes:

```
>>> df1 = pd.DataFrame([[1, np.nan]])
>>> df2 = pd.DataFrame([[3, 4]])
>>> df1.combine_first(df2)
   0    1
0  1  4.0
```

DataFrame.combine [Perform series-wise operation on two DataFrames] using a given function

compound(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

compounded : Series or DataFrame (if level specified)

consolidate(*inplace=False*)

Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

convert_objects(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Return a fixed frequency timedelta index,

with day as the default.

converted : same as input object

copy(*deep=True*)

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

deep [bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

copy [Series, DataFrame or Panel] Object type matches caller.

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
```

(continues on next page)

(continued from previous page)

```
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

corr (*method='pearson'*, *min_periods=1*)

Compute pairwise correlation of columns, excluding NA/null values

method [{'pearson', 'kendall', 'spearman'}]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation

y : DataFrame

corrwith (*other*, *axis=0*, *drop=False*)

Compute pairwise correlation between rows or columns of two DataFrame objects.

other : DataFrame, Series axis : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' to compute column-wise, 1 or 'columns' for row-wise

drop [boolean, default False] Drop missing indices from result, default returns union of all

correls : Series

count (*axis=0*, *level=None*, *numeric_only=False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, and optionally *numpy.inf* (depending on *pandas.options.mode.use_inf_as_na*) are considered NA.

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each **row**.

level [int or str, optional] If the axis is a *MultiIndex* (hierarchical), count along a particular *level*, collapsing into a *DataFrame*. A *str* specifies the level name.

numeric_only [boolean, default False] Include only *float*, *int* or *boolean* data.

Series or DataFrame For each column/row the number of non-NA/null entries. If *level* is specified returns a *DataFrame*.

Series.count: number of non-NA elements in a Series DataFrame.shape: number of DataFrame rows and columns (including NA

elements)

DataFrame.isna: boolean same-sized DataFrame showing places of NA elements

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                    ["John", "Myla", None, "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> df
   Person   Age  Single
0    John  24.0   False
1    Myla   NaN    True
2    None  21.0    True
3    John  33.0    True
4    Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    4
Age       4
Single    5
dtype: int64
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    2
3    3
4    3
dtype: int64
```

Counts for one level of a *MultiIndex*:

```
>>> df.set_index(["Person", "Single"]).count(level="Person")
        Age
Person
John      2
Myla      1
```

cov (*min_periods=None*)

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as `NaN`.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result.

DataFrame The covariance matrix of the series of the DataFrame.

pandas.Series.cov : compute covariance with another Series pandas.core.window.EWM.cov: expoential weighted sample covariance pandas.core.window.Expanding.cov : expanding sample covariance pandas.core.window.Rolling.cov : rolling sample covariance

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                   columns=['dogs', 'cats'])
>>> df.cov()
          dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
          a         b         c         d         e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
```

(continues on next page)

(continued from previous page)

```
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
          a         b         c
a  0.316741       NaN -0.150812
b       NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

cummax (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummax : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.max [Similar functionality] but ignores `NaN` values.

DataFrame.max [Return the maximum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cummin (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummin : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
```

(continues on next page)

(continued from previous page)

```
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.min [Similar functionality] but ignores `NaN` values.

DataFrame.min [Return the minimum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumprod (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumprod : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.prod [Similar functionality] but ignores `NaN` values.

DataFrame.prod [Return the product over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumsum(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumsum : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.sum [Similar functionality] but ignores `NaN` values.

DataFrame.sum [Return the sum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

`describe` (*percentiles=None*, *include=None*, *exclude=None*)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

summary: Series/DataFrame of summary statistics

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                       3
unique                      2
top       2010-01-01 00:00:00
freq                        2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                     'numeric': [1, 2, 3],
...                     'categorical': pd.Categorical(['d','e','f'])
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
        categorical  numeric object
count             3      3.0      3
unique            3      NaN      3
top               f      NaN      c
freq              1      NaN      1
mean            NaN      2.0    NaN
std             NaN      1.0    NaN
min             NaN      1.0    NaN
25%             NaN      1.5    NaN
50%             NaN      2.0    NaN
75%             NaN      2.5    NaN
max             NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
        categorical  numeric
count            3      3.0
```

(continues on next page)

(continued from previous page)

```
unique            3      NaN
top               f      NaN
freq              1      NaN
mean            NaN      2.0
std             NaN      1.0
min             NaN      1.0
25%             NaN      1.5
50%             NaN      2.0
75%             NaN      2.5
max             NaN      3.0
```

DataFrame.count DataFrame.max DataFrame.min DataFrame.mean DataFrame.std DataFrame.select_dtypes

diff (*periods=1*, *axis=0*)

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

periods [int, default 1] Periods to shift for calculating difference, accepts negative values.

axis [{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

New in version 0.16.1..

diffed : DataFrame

Series.diff: First discrete difference for a Series. DataFrame.pct_change: Percent change over given number of periods. DataFrame.shift: Shift index by desired number of periods with an

optional time freq.

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a    b     c
0 NaN  0.0   0.0
1 NaN -1.0   3.0
2 NaN -1.0   7.0
3 NaN -1.0  13.0
4 NaN  0.0  20.0
5 NaN  2.0  28.0
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

div(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

divide(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

dot (*other*)

Matrix multiplication with DataFrame or Series objects. Can also be called using *self @ other* in Python >= 3.5.

other : DataFrame or Series

dot_product : DataFrame or Series

drop (*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

labels [single label or list-like] Index or column labels to drop.

axis [{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index, columns [single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

New in version 0.21.0.

level [int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace [bool, default False] If True, do operation inplace and return None.

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

dropped : pandas.DataFrame

DataFrame.loc : Label-location based indexer for selection by label. DataFrame.dropna : Return DataFrame with labels on given axis omitted

where (all or any) data are missing

DataFrame.drop_duplicates [Return DataFrame with duplicate rows] removed, optionally only considering certain columns

Series.drop : Return Series with specified index labels removed.

KeyError If none of the labels are found in the selected axis

```
>>> df = pd.DataFrame(np.arange(12).reshape(3,4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                              [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3,0.2]])
>>> df
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
lama    speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
```

(continues on next page)

(continued from previous page)

```
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

drop_duplicates (*subset=None*, *keep='first'*, *inplace=False*)

Return DataFrame with duplicate rows removed, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Drop duplicates except for the first occurrence.
- `last` : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

inplace [boolean, default False] Whether to drop duplicates in place or to return a copy

deduplicated : DataFrame

dropna (*axis=0*, *how='any'*, *thresh=None*, *subset=None*, *inplace=False*)

Remove missing values.

See the User Guide for more on which values are considered missing, and how to work with missing data.

axis [{0 or 'index', 1 or 'columns'}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Deprecated since version 0.23.0:: Pass tuple or list to drop on multiple

axes.

how [{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

thresh [int, optional] Require that many non-NA values.

subset [array-like, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace [bool, default False] If True, do operation inplace and return None.

DataFrame DataFrame with NA entries dropped from it.

DataFrame.isna: Indicate missing values. DataFrame.notna : Indicate existing (non-missing) values. DataFrame.fillna : Replace missing values. Series.dropna : Drop missing values. Index.dropna : Drop missing indices.

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'born'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type of each column.

pandas.DataFrame.ftypes : dtype and sparsity information.

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
string              object
dtype: object
```

duplicated(*subset=None*, *keep='first'*)

Return boolean Series denoting duplicate rows, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Mark duplicates as `True` except for the first occurrence.
- `last` : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

duplicated : Series

empty

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

bool If DataFrame is empty, return True, if not return False.

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
```

(continues on next page)

(continued from previous page)

```
>>> df.dropna().empty
True
```

pandas.Series.dropna pandas.DataFrame.dropna

eq(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods eq

equals(*other*)
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

eval(*expr*, *inplace=False*, ***kwargs*)
Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

expr [str] The expression string to evaluate.

inplace [bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

New in version 0.18.0..

kwargs [dict] See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

ndarray, scalar, or pandas object The result of the evaluation.

DataFrame.query [Evaluates a boolean expression to query the columns] of a frame.

DataFrame.assign [Can evaluate an expression or function to create new] values for a column.

pandas.eval [Evaluate a Python expression as a string using various] backends.

For more details see the API documentation for `eval()`. For detailed examples see enhancing performance with eval.

```
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
>>> df.eval('A + B')
0    11
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
```

Use `inplace=True` to modify the original DataFrame.

```
>>> df.eval('C = A + B', inplace=True)
>>> df
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
```

ewm(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*)
Provides exponential weighted functions

New in version 0.18.0.

com [float, optional] Specify decay in terms of center of mass, $\alpha = 1/(1 + com),$ for $com \geq 0$

span [float, optional] Specify decay in terms of span, $\alpha = 2/(span + 1),$ for $span \geq 1$

halflife [float, optional] Specify decay in terms of half-life, $\alpha = 1 - exp(log(0.5)/halflife),$ for $halflife > 0$

alpha [float, optional] Specify smoothing factor α directly, $0 < \alpha \leq 1$

New in version 0.18.0.

min_periods [int, default 0] Minimum number of observations in window required to have a value (otherwise result is NA).

adjust [boolean, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

ignore_na [boolean, default False] Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), . . . , 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

rolling : Provides rolling window calculations expanding : Provides expanding transformations.

expanding (*min_periods=1*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

min_periods [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

center [boolean, default False] Set the labels at the center of the window.

axis : int or string, default 0

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

rolling : Provides rolling window calculations ewm : Provides exponential weighted functions

ffill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='ffill')`

fillna(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)
Fill NA/NaN values using the specified method

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis : {0 or 'index', 1 or 'columns'} inplace : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

interpolate : Fill NaN values using interpolation. reindex, asfreq

filled : DataFrame

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 2.0 1
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

filter(*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

items [list-like] List of info axis to restrict to (must not all be present)

like [string] Keep info axis where "arg in col == True"

regex [string (regular expression)] Keep info axis with re.search(regex, col) == True

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame

same type as input object

```
>>> df
one  two  three
mouse     1    2      3
rabbit    4    5      6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
one  two  three
rabbit    4    5      6
```

pandas.DataFrame.loc

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

filter_result(*expressions*)

Filters a result data frame based on a specified expression consisting of a list of triple with (method_name, comparator, threshold). The expression is applied to each row. If any of the columns fulfill the criteria the row remains.

Parameters **expressions** (*list((str, comparator, float))*) – A list of triples consisting of (method_name, comparator, threshold)

Returns A new filtered result object

Return type *CleavageSitePredictionResult*

first(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

subset : type of caller

last : Select final periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

first_valid_index()

Return index for first non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

floordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rfloordiv

classmethod from_csv(*path*, *header=0*, *sep=','*, *','*, *index_col=0*, *parse_dates=True*, *encoding=None*, *tupleize_cols=None*, *infer_datetime_format=False*)
Read CSV file.

Deprecated since version 0.21.0: Use `pandas.read_csv()` instead.

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a DataFrame of time series data.

This method only differs from the preferred `pandas.read_csv()` in some defaults:

- *index_col* is `0` instead of `None` (take first column as index by default)
- *parse_dates* is `True` instead of `False` (try parsing the index as datetime by default)

So a `pd.DataFrame.from_csv(path)` can be replaced by `pd.read_csv(path, index_col=0, parse_dates=True)`.

path : string file path or file handle / StringIO header : int, default 0

Row to use as header (skip prior rows)

sep [string, default ','] Field delimiter

index_col [int or sequence, default 0] Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

parse_dates [boolean, default True] Parse dates. Different default from read_table

tupleize_cols [boolean, default False] write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

infer_datetime_format: boolean, default False If True and *parse_dates* is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

pandas.read_csv

y : DataFrame

classmethod from_dict(*data*, *orient='columns'*, *dtype=None*, *columns=None*)
Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

data [dict] Of the form {field : array-like} or {field : dict}.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'.

dtype [dtype, default None] Data type to force, otherwise infer.

columns [list, default None] Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'`.

New in version 0.23.0.

pandas.DataFrame

DataFrame.from_records [DataFrame from ndarray (structured] dtype), list of tuples, dict, or DataFrame

DataFrame : DataFrame object creation using constructor

By default the keys of the dict become the DataFrame columns:

```
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Specify `orient='index'` to create the DataFrame using dictionary keys as rows:

```
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
       0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                        columns=['A', 'B', 'C', 'D'])
       A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
```

classmethod from_items(*items*, *columns=None*, *orient='columns'*)
Construct a dataframe from a list of tuples

Deprecated since version 0.23.0: *from_items* is deprecated and will be removed in a future version. Use `DataFrame.from_dict(dict(items))` instead. `DataFrame.from_dict(OrderedDict(items))` may be used to preserve the key order.

Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the columns, but depends on the specified orientation). The values should be arrays or Series.

items [sequence of (key, value) pairs] Values should be arrays or Series.

columns [sequence of column labels, optional] Must be passed if orient='index'.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the input correspond to column labels, pass 'columns' (default). Otherwise if the keys correspond to the index, pass 'index'.

frame : DataFrame

classmethod from_records(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float=False*, *nrows=None*)

Convert structured or record ndarray to DataFrame

data : ndarray (structured dtype), list of tuples, dict, or DataFrame index : string, list of fields, array-like

Field of array to use as the index, alternately a specific set of input labels to use

exclude [sequence, default None] Columns or fields to exclude

columns [sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

coerce_float [boolean, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

df : DataFrame

ftypes

Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type and indication of sparse/dense of each column.

pandas.DataFrame.dtypes: Series with just dtype information. pandas.SparseDataFrame : Container for sparse tabular data.

Sparse data should have the same dtypes as its dense representation.

```
>>> import numpy as np
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).ftypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).ftypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

ge(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods ge

get(*key*, *default=None*)
Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

key : object

value : type of items contained in object

get_dtype_counts()
Return counts of unique dtypes in this object.

dtype [Series] Series with the count of columns with each dtype.

dtypes : Return the dtypes in this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

get_ftype_counts()
Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

ftypes [Return ftypes (indication of sparse/dense and dtype) in] this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

get_value(*index*, *col*, *takeable=False*)
Quickly retrieve single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label takeable : interpret the index/col as indexers, default False

value : scalar value

get_values()

Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a *pandas.SparseArray*, the data are first converted to a dense representation.

numpy.ndarray Numpy representation of DataFrame

values : Numpy representation of DataFrame. pandas.SparseArray : Container for sparse data.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a      b    c
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
     a    c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

groupby(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=False*, *observed=False*, ***kwargs*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

by [mapping, function, label, or list of labels] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted a (single) key.

axis : int, default 0 level : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

as_index [boolean, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort [boolean, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.

group_keys [boolean, default True] When calling apply, add group keys to index to identify pieces

squeeze [boolean, default False] reduce the dimensionality of the return type if possible, otherwise return a consistent type

observed [boolean, default False] This only applies if any of the groupers are Categoricals If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

GroupBy object

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

See the user guide for more.

resample [Convenience method for frequency conversion and resampling] of time series.

gt (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods gt

head (*n=5*)
Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

n [int, default 5] Number of rows to select.

obj_head [type of caller] The first *n* rows of the caller object.

pandas.DataFrame.tail: Returns the last *n* rows.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
```

(continues on next page)

(continued from previous page)

```
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

hist(*column=None*, *by=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *ax=None*, *sharex=False*, *sharey=False*, *figsize=None*, *layout=None*, *bins=10*, ***kwds*)
Make a histogram of the DataFrame's.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

data [DataFrame] The pandas object holding the data.

column [string or sequence] If passed, will be used to limit data to a subset of columns.

by [object, optional] If passed, then used to form histograms for separate groups.

grid [boolean, default True] Whether to show axis grid lines.

xlabelsize [int, default None] If specified changes the x-axis label size.

xrot [float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize [int, default None] If specified changes the y-axis label size.

yrot [float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

ax [Matplotlib axes object, default None] The axes to plot the histogram on.

sharex [boolean, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey [boolean, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize [tuple] The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

layout [tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

axes : matplotlib.AxesSubplot or numpy.ndarray of them

matplotlib.pyplot.hist : Plot a histogram using matplotlib.

iat

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.at : Access a single value for a row/column label pair DataFrame.loc : Access a group of rows and columns by label(s) DataFrame.iloc : Access a group of rows and columns by integer position(s)

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

IndexError When integer position is out of bounds

idxmax (*axis=0*, *skipna=True*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmax : Series

This method is the DataFrame version of `ndarray.argmax`.

Series.idxmax

idxmin (*axis=0*, *skipna=True*)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmin : Series

This method is the DataFrame version of `ndarray.argmin`.

Series.idxmin

iloc

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at Selection by Position

index

The index (row labels) of the DataFrame.

infer_objects()

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to numeric typeR

converted : same type as input object

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

info(*verbose=None*, *buf=None*, *max_cols=None*, *memory_usage=None*, *null_counts=None*)
Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

verbose [bool, optional] Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

buf [writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

max_cols [int, optional] When to switch from the verbose to the truncated output. If the DataFrame has more than *max_cols* columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

memory_usage [bool, str, optional] Specifies whether total memory usage of the DataFrame elements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources.

null_counts [bool, optional] Whether to show the non-null counts. By default, this is shown only if the frame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

None This method prints a summary of a DataFrame and returns None.

DataFrame.describe: Generate descriptive statistics of DataFrame columns.

DataFrame.memory_usage: Memory usage of DataFrame columns.

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                    "float_col": float_values})
>>> df
   int_col text_col  float_col
0        1    alpha       0.00
1        2     beta       0.25
2        3    gamma       0.50
3        4    delta       0.75
4        5  epsilon       1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
int_col      5 non-null int64
```

(continues on next page)

(continued from previous page)

```
text_col     5 non-null object
float_col    5 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w", encoding="utf-8") as f:
...     f.write(s)
260
```

The *memory_usage* parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 22.9+ MB
```

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 188.8 MB
```

insert (*loc*, *column*, *value*, *allow_duplicates=False*)

Insert column into DataFrame at specified location.

Raises a ValueError if *column* is already contained in the DataFrame, unless *allow_duplicates* is set to

True.

loc [int] Insertion index. Must verify 0 <= loc <= len(columns)

column [string, number, or hashable object] label of the inserted column

value : int, Series, or array-like allow_duplicates : bool, optional

interpolate(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

method [{'linear', 'time', 'index', 'values', 'nearest', 'zero',]

'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to `scipy.interpolate.interp1d`. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
- 'from_derivatives' refers to BPoly.from_derivatives which replaces 'piecewise_polynomial' interpolation method in scipy 0.18

New in version 0.18.1: Added support for the 'akima' method Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in scipy 0.18; backwards-compatible with scipy < 0.18

axis [{0, 1}, default 0]

- 0: fill column-by-column
- 1: fill row-by-row

limit [int, default None.] Maximum number of consecutive NaNs to fill. Must be greater than 0.

limit_direction : {'forward', 'backward', 'both'}, default 'forward' limit_area : {'inside', 'outside'}, default None

- None: (default) no fill restriction
- 'inside' Only fill NaNs surrounded by valid values (interpolate).
- 'outside' Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

inplace [bool, default False] Update the NDFrame in place if possible.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

kwargs : keyword arguments to pass on to the interpolating function.

Series or DataFrame of same shape interpolated at the NaNs

reindex, replace, fillna

Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

is_copy

isin(*values*)

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

values [iterable, Series, DataFrame or dictionary] The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dictionary, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

DataFrame of booleans

When `values` is a list:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
       A      B
0   True   True
1  False  False
2   True  False
```

When `values` is a dict:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
       A      B
0   True  False  # Note that B didn't match the 1 here.
1  False   True
2   True   True
```

When `values` is a Series or DataFrame:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
       A      B
0   True  False
1  False  False  # Column A in `other` has a 3, but not at index 1.
2   True   True
```

isna()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

isnull()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

items()

Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iteritems()

Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iterrows()

Iterate over DataFrame rows as (index, Series) pairs.

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
```

(continues on next page)

(continued from previous page)

```
>>> print(df['int'].dtype)
int64
```

To preserve dtypes while iterating over the rows, it is better to use *itertuples()* which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

it [generator] A generator that iterates over the rows of the frame.

itertuples : Iterate over DataFrame rows as namedtuples of the values. iteritems : Iterate over (column name, Series) pairs.

itertuples(*index=True*, *name='Pandas'*)

Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.

index [boolean, default True] If True, return the index as the first element of the tuple.

name [string, default "Pandas"] The name of the returned namedtuples or None to return regular tuples.

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. iteritems : Iterate over (column name, Series) pairs.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [0.1, 0.2]},
                      index=['a', 'b'])
>>> df
   col1  col2
a     1   0.1
b     2   0.2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='a', col1=1, col2=0.10000000000000001)
Pandas(Index='b', col1=2, col2=0.20000000000000001)
```

ix

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at Advanced Indexing.

join(*other*, *on=None*, *how='left'*, *lsuffix=''*, *rsuffix=''*, *sort=False*)

Join columns with other DataFrame either on index or on a key column. Efficiently Join multiple DataFrame objects by index at once by passing a list.

other [DataFrame, Series with name field set, or list of DataFrame] Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame

on [name, tuple/list of names, or array-like] Column or index level name(s) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

how [{'left', 'right', 'outer', 'inner'}, default: 'left'] How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use other frame's index
- outer: form union of calling frame's index (or column if on is specified) with other frame's index, and sort it lexicographically
- inner: form intersection of calling frame's index (or column if on is specified) with other frame's index, preserving the order of the calling's one

lsuffix [string] Suffix to use from left frame's overlapping columns

rsuffix [string] Suffix to use from right frame's overlapping columns

sort [boolean, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword)

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

Support for specifying index levels as the *on* parameter was added in version 0.23.0

```
>>> caller = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                        'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> caller
    A key
0  A0  K0
1  A1  K1
2  A2  K2
3  A3  K3
4  A4  K4
5  A5  K5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
    B key
0  B0  K0
1  B1  K1
2  B2  K2
```

Join DataFrames using their indexes.

```
>>> caller.join(other, lsuffix='_caller', rsuffix='_other')
```

```
>>>     A key_caller    B key_other
    0  A0         K0   B0        K0
    1  A1         K1   B1        K1
```

(continues on next page)

(continued from previous page)

```
    2  A2          K2   B2         K2
    3  A3          K3  NaN        NaN
    4  A4          K4  NaN        NaN
    5  A5          K5  NaN        NaN
```

If we want to join using the key columns, we need to set key to be the index in both caller and other. The joined DataFrame will have key as its index.

```
>>> caller.set_index('key').join(other.set_index('key'))
```

```
>>>       A    B
    key
    K0   A0   B0
    K1   A1   B1
    K2   A2   B2
    K3   A3  NaN
    K4   A4  NaN
    K5   A5  NaN
```

Another option to join using the key columns is to use the on parameter. DataFrame.join always uses other's index but we can use any column in the caller. This method preserves the original caller's index in the result.

```
>>> caller.join(other.set_index('key'), on='key')
```

```
>>>     A key    B
    0  A0  K0   B0
    1  A1  K1   B1
    2  A2  K2   B2
    3  A3  K3  NaN
    4  A4  K4  NaN
    5  A5  K5  NaN
```

DataFrame.merge : For column(s)-on-columns(s) operations

joined : DataFrame

keys()

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

kurt(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

kurtosis (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

last (*offset*)
Convenience method for subsetting final periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

subset : type of caller

first : Select initial periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

last_valid_index ()
Return index for last non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

le (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods le

loc
Access a group of rows and columns by label(s) or a boolean array.

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

> **Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at Selection by Label

DataFrame.at : Access a single value for a row/column label pair DataFrame.iloc : Access group of rows and columns by integer position(s) DataFrame.xs : Returns a cross-section (row(s) or column(s)) from the

Series/DataFrame.

Series.loc : Access group of values using labels

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=['cobra', 'viper', 'sidewinder'],
...       columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
```

(continues on next page)

(continued from previous page)

```
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

KeyError: when any items are not found

lookup(*row_labels*, *col_labels*)

Label-based "fancy indexing" function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

row_labels [sequence] The row labels to use for lookup

col_labels [sequence] The column labels to use for lookup

Akin to:

```
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

values [ndarray] The found values

lt(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods lt

mad(*axis=None*, *skipna=None*, *level=None*)
Return the mean absolute deviation of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mad : Series or DataFrame (if level specified)

mask(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

`DataFrame.where()`

max(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the maximum of the values in the object. If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

max : Series or DataFrame (if level specified)

mean(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mean : Series or DataFrame (if level specified)

median(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

median : Series or DataFrame (if level specified)

melt(*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *col_level=None*)

"Unpivots" a DataFrame from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

New in version 0.20.0.

frame : DataFrame id_vars : tuple, list, or ndarray, optional

Column(s) to use as identifier variables.

value_vars [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name [scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name [scalar, default 'value'] Name to use for the 'value' column.

col_level [int or string, optional] If columns are a MultiIndex then use this level to melt.

melt pivot_table DataFrame.pivot

```
>>> import pandas as pd
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> df.melt(id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

memory_usage(*index=True*, *deep=False*)

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

index [bool, default True] Specifies whether to include the memory usage of the DataFrame's index in returned Series. If `index=True` the memory usage of the index the first item in the output.

deep [bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

sizes [Series] A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

numpy.ndarray.nbytes [Total bytes consumed by the elements of an] ndarray.

Series.memory_usage : Bytes consumed by a Series. pandas.Categorical : Memory-efficient array for string values with

many repeated values.

DataFrame.info : Concise summary of a DataFrame.

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64  complex128 object  bool
0      1      1.0      (1+0j)      1  True
1      1      1.0      (1+0j)      1  True
2      1      1.0      (1+0j)      1  True
3      1      1.0      (1+0j)      1  True
4      1      1.0      (1+0j)      1  True
```

```
>>> df.memory_usage()
Index            80
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64         40000
float64       40000
complex128    80000
object        40000
```

(continues on next page)

(continued from previous page)

```
bool           5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index             80
int64          40000
float64        40000
complex128     80000
object        160000
bool            5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5168
```

merge(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *sort=False*, *suffixes=('_x', '_y')*, *copy=True*, *indicator=False*, *validate=None*)

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

right : DataFrame how : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys

on [label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [boolean, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

right_index [boolean, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index

sort [boolean, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword)

suffixes [2-length sequence (tuple, list, ...)] Suffix to apply to overlapping column names in the left and right side, respectively

copy [boolean, default True] If False, do not copy data unnecessarily

indicator [boolean or string, default False] If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in 'left' DataFrame, "right_only" for observations whose merge key only appears in 'right' DataFrame, and "both" if the observation's merge key is found in both.

validate [string, default None] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

New in version 0.21.0.

Support for specifying index levels as the *on*, *left_on*, and *right_on* parameters was added in version 0.23.0

```
>>> A              >>> B
    lkey value         rkey value
0   foo  1         0   foo  5
1   bar  2         1   bar  6
2   baz  3         2   qux  7
3   foo  4         3   bar  8
```

```
>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0  foo   1        foo   5
1  foo   4        foo   5
2  bar   2        bar   6
3  bar   2        bar   8
4  baz   3        NaN   NaN
5  NaN   NaN      qux   7
```

merged [DataFrame] The output type will the be same as 'left', if it is a subclass of DataFrame.

merge_ordered merge_asof DataFrame.join

merge_results(*others*)

Merges results of type *CleavageSitePredictionResult* and returns the merged result

Parameters **others** (list(*CleavageSitePredictionResult*) or *CleavageSitePredictionResult*) – A (list of) *CleavageSitePredictionResult* object(s)

Returns A new merged *CleavageSitePredictionResult* object

Return type *CleavageSitePredictionResult*

min(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the minimum of the values in the object. If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min : Series or DataFrame (if level specified)

mod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmod

mode(*axis=0*, *numeric_only=False*)

Gets the mode(s) of each element along the axis selected. Adds a row for each mode per label, fills in gaps with nan.

Note that there could be multiple values returned for the selected axis (when more than one item share the maximum frequency), which is the reason why a dataframe is returned. If you want to impute missing values with the mode in a dataframe `df`, you can just do this: `df.fillna(df.mode().iloc[0])`

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index' : get mode of each column
- 1 or 'columns' : get mode of each row

numeric_only [boolean, default False] if True, only apply to numeric columns

modes : DataFrame (sorted)

```
>>> df = pd.DataFrame({'A': [1, 2, 1, 2, 1, 2, 3]})
>>> df.mode()
   A
0  1
1  2
```

mul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

multiply(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

ndim

Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

ndarray.ndim

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

ne(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods ne

nlargest (*n*, *columns*, *keep='first'*)

Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant.

n [int] Number of rows to return.

columns [label or list of labels] Column label(s) to order by.

keep [{'first', 'last'}, default 'first'] Where there are duplicate values:

- *first* : prioritize the first occurrence(s)
- *last* : prioritize the last occurrence(s)

DataFrame The first *n* rows ordered by the given columns in descending order.

DataFrame.nsmallest [Return the first *n* rows ordered by *columns* in] ascending order.

DataFrame.sort_values : Sort DataFrame by the values DataFrame.head : Return the first *n* rows without re-ordering.

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 10, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df
    a  b    c
0   1  a  1.0
1  10  b  2.0
2   8  d  NaN
3  10  c  3.0
4  -1  e  4.0
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column "a".

```
>>> df.nlargest(3, 'a')
    a  b    c
1  10  b  2.0
3  10  c  3.0
2   8  d  NaN
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'a', keep='last')
    a  b    c
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

To order by the largest values in column "a" and then "c", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['a', 'c'])
    a  b    c
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

Attempting to use `nlargest` on non-numeric dtypes will raise a `TypeError`:

```
>>> df.nlargest(3, 'b')
Traceback (most recent call last):
TypeError: Column 'b' has dtype object, cannot use method 'nlargest'
```

notna()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
```

(continues on next page)

(continued from previous page)

```
2    False
dtype: bool
```

notnull()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

nsmallest(*n*, *columns*, *keep='first'*)

Get the rows of a DataFrame sorted by the *n* smallest values of *columns*.

n [int] Number of items to retrieve

columns [list or str] Column name or names to order by

keep [{'first', 'last'}, default 'first'] Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

DataFrame

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 11, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nsmallest(3, 'a')
   a  b   c
4 -1  e   4
0  1  a   1
2  8  d NaN
```

nunique (*axis=0*, *dropna=True*)

Return Series with number of distinct observations over requested axis.

New in version 0.20.0.

axis : {0 or 'index', 1 or 'columns'}, default 0 dropna : boolean, default True

Don't include NaN in the counts.

nunique : Series

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 1, 1]})
>>> df.nunique()
A    3
B    1
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
```

pct_change (*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs** Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

chg [Series or DataFrame] The same type as the calling object.

Series.diff : Compute the difference of two elements in a Series. DataFrame.diff : Compute the difference of two elements in a DataFrame. Series.shift : Shift the index by some number of periods. DataFrame.shift : Shift the index by some number of periods.

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...      'FR': [4.0405, 4.0963, 4.3149],
...      'GR': [1.7246, 1.7482, 1.8519],
...      'IT': [804.74, 810.01, 860.13]},
...      index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                  FR        GR        IT
1980-01-01       NaN       NaN       NaN
1980-02-01  0.013810  0.013684  0.006549
1980-03-01  0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...      '2016': [1769950, 30586265],
...      '2015': [1500923, 40912316],
...      '2014': [1371819, 41403351]},
...      index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pipe (*func*, **args*, ***kwargs*)

Apply func(self, *args, **kwargs)

func [function] function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the NDFrame.

args [iterable, optional] positional arguments passed into `func`.

kwargs [mapping, optional] a dictionary of keyword arguments passed into `func`.

object : the return type of `func`.

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pandas.DataFrame.apply pandas.DataFrame.applymap pandas.Series.map

pivot (*index=None*, *columns=None*, *values=None*)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the User Guide for more on reshaping.

index [string or object, optional] Column to use to make new frame's index. If None, uses existing index.

columns [string or object] Column to use to make new frame's columns.

values [string, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Changed in version 0.23.0: Also accept list of column names.

DataFrame Returns reshaped DataFrame.

ValueError: When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

DataFrame.pivot_table [generalization of pivot that can handle] duplicate values for one index/column pair.

DataFrame.unstack [pivot based on the index values instead of a] column.

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
...                    "baz": [1, 2, 3, 4]})
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

pivot_table(*values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*, *dropna=True*, *margins_name='All'*)

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame

values : column to aggregate, optional index : column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

columns [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

aggfunc [function, list of functions, dict, default numpy.mean] If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions

fill_value [scalar, default None] Value to replace missing values with

margins [boolean, default False] Add all row / columns (e.g. for subtotal / grand totals)

dropna [boolean, default True] Do not include columns whose entries are all NaN

margins_name [string, default 'All'] Name of the row / column that will contain the totals when margins is True.

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
>>> df
     A    B      C  D
0  foo  one  small  1
1  foo  one  large  2
```

(continues on next page)

(continued from previous page)

```
2  foo  one  large  2
3  foo  two  small  3
4  foo  two  small  3
5  bar  one  large  4
6  bar  one  small  5
7  bar  two  small  6
8  bar  two  large  7
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': [min, max, np.mean]})
>>> table
                D   E
             mean max median min
A   C
bar large  5.500000  16   14.5  13
    small  5.500000  15   14.5  14
foo large  2.000000  10    9.5   9
    small  2.333333  12   11.0   8
```

table : DataFrame

DataFrame.pivot [pivot without aggregation that can handle] non-numeric data

plot

alias of `pandas.plotting._core.FramePlotMethods`

pop (*item*)

Return item and drop from frame. Raise KeyError if not found.

item [str] Column label to be popped

popped : Series

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
```

(continues on next page)

(continued from previous page)

```
...                    ('monkey', 'mammal', np.nan)],
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

pow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rpow

prod(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

product (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

quantile(*q=0.5*, *axis=0*, *numeric_only=True*, *interpolation='linear'*)

Return values at the given quantile over requested axis, a la numpy.percentile.

q [float or array-like, default 0.5 (50% quantile)] 0 <= q <= 1, the quantile(s) to compute

axis [{0, 1, 'index', 'columns'} (default 0)] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

numeric_only [boolean, default True] If False, the quantile of datetime and timedelta data will be computed as well

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.18.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

quantiles : Series or DataFrame

- If `q` is an array, a DataFrame will be returned where the index is `q`, the columns are the columns of self, and the values are the quantiles.
- If `q` is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                      columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
                       'B': [pd.Timestamp('2010'),
                             pd.Timestamp('2011')],
                       'C': [pd.Timedelta('1 days'),
                             pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
```

(continues on next page)

(continued from previous page)

```
C        1 days 12:00:00
Name: 0.5, dtype: object
```

pandas.core.window.Rolling.quantile

query(*expr*, *inplace=False*, ***kwargs*)

Query the columns of a frame with a boolean expression.

expr [string] The query string to evaluate. You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

inplace [bool] Whether the query should modify the data in place or return a modified copy

New in version 0.18.0.

kwargs [dict] See the documentation for `pandas.eval()` for complete details on the keyword arguments accepted by `DataFrame.query()`.

q : DataFrame

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level `pandas.eval()` function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the `query` documentation in indexing.

pandas.eval DataFrame.eval

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = pd.DataFrame(randn(10, 2), columns=list('ab'))
>>> df.query('a > b')
>>> df[df.a > df.b]  # same result as the previous expression
```

radd(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.add

rank (*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)
Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values

axis [{0 or 'index', 1 or 'columns'}, default 0] index to direct ranking

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

ranks : same type as caller

rdiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

reindex(***kwargs*)
Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like, optional] New labels / index to conform the axis specified by 'axis' to.

index, columns [array-like, optional (should be specified using keywords)] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis [int or str, optional] Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...       'http_status': [200,200,404,404,301],
...       'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...        index=index)
>>> df
            http_status  response_time
Firefox             200           0.04
Chrome              200           0.02
Safari              404           0.07
IE10                404           0.08
Konqueror           301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
```

(continues on next page)

(continued from previous page)

```
IE10                 404          0.08
Chrome               200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to backpropagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the `method` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29     100
2009-12-30     100
2009-12-31     100
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

Please note that the `NaN` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the `NaN` values present in the original dataframe, use the `fillna()` method.

See the user guide for more.

reindexed : DataFrame

`reindex_axis`(*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=nan*)
Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis : {0 or 'index', 1 or 'columns'} method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional

Method to use for filling holes in reindexed DataFrame:

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

reindex, reindex_like

reindexed : DataFrame

reindex_like(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)
Return an object with matching indices to myself.

other : Object method : string or None copy : boolean, default True limit : int, default None

Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

reindexed : same as input

rename(***kwargs*)
Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the user guide for more.

mapper, index, columns [dict-like or function, optional] dict-like or functions transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

axis [int or str, optional] Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

copy [boolean, default True] Also copy underlying data

inplace [boolean, default False] Whether to return a new DataFrame. If True then value of copy is ignored.

level [int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

renamed : DataFrame

pandas.DataFrame.rename_axis

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
```

(continues on next page)

(continued from previous page)

```
1  2  5
2  3  6
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

rename_axis(*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter the name of the index or columns.

mapper [scalar, list-like, optional] Value to set as the axis name attribute.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis.

copy [boolean, default True] Also copy underlying data.

inplace [boolean, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

renamed [Series, DataFrame, or None] The same type as the caller or None if *inplace* is True.

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

pandas.Series.rename : Alter Series index labels or name pandas.DataFrame.rename : Alter DataFrame index labels or name pandas.Index.rename : Set new names on index

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s.rename_axis("foo")
foo
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")
     A  B
foo
0    1  4
1    2  5
2    3  6
```

```
>>> df.rename_axis("bar", axis="columns")
bar  A  B
0    1  4
1    2  5
2    3  6
```

reorder_levels (*order*, *axis=0*)

Rearrange index levels using input order. May not drop or duplicate levels

order [list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).

axis [int] Where to reorder levels.

type of caller (new object)

replace (*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)

Replace values given in *to_replace* with *value*.

Values of the DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

to_replace [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be `None` to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:

- This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

value [scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace [boolean, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

regex [bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method [{'pad', 'ffill', 'bfill', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

DataFrame.fillna : Fill NA values DataFrame.where : Replace values based on boolean condition Series.str.replace : Simple string replacement.

DataFrame Object after replacement.

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Scalar ‘to_replace‘ and ‘value‘

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like ‘to_replace‘

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
2    3
3    3
4    4
dtype: int64
```

dict-like ‘to_replace‘

```
>>> df.replace({0: 10, 1: 100})
     A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
     A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
     A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression 'to_replace'

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
      A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$':'new', 'foo':'xyz'})
      A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
      A    B
0   new  abc
1   new  new
2  bait  xyz
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
```

(continues on next page)

(continued from previous page)

```
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the pecularities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value=None` and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

resample(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

rule [string] the offset string or object representing target conversion

axis : int, optional, default 0 closed : {'right', 'left'}

Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label [{'right', 'left'}] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention [{'start', 'end', 's', 'e'}] For PeriodIndex only, controls whether to use the start or end of *rule*

kind: {'timestamp', 'period'}, optional Pass 'timestamp' to convert the resulting index to a `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`. By default the input representation is retained.

loffset [timedelta] Adjust the resampled time labels

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

on [string, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [string or int, optional] For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

Resampler object

See the user guide for more.

To learn more about the offset strings, please see this link.

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5] #select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like)+5
```

```
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))
>>> s
2012    1
```

(continues on next page)

(continued from previous page)

```
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using 'start' *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
                     a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                       columns=['a', 'b', 'c', 'd'],
                       index=pd.MultiIndex.from_product([time, [1, 2]])
                       )
>>> df2.resample('3T', level=0).sum()
                     a  b   c   d
2000-01-01 00:00:00  0  6  12  18
2000-01-01 00:03:00  0  4   8  12
```

groupby : Group by mapping, function, label, or list of labels.

reset_index(*level=None*, *drop=False*, *inplace=False*, *col_level=0*, *col_fill=''*)

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level_0' (if 'index' is already taken) will be used.

level [int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default

drop [boolean, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

col_level [int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill [object, default ''] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

resetted : DataFrame

```
>>> df = pd.DataFrame([('bird',    389.0),
...                    ('bird',     24.0),
...                    ('mammal',   80.5),
...                    ('mammal', np.nan)],
...                   index=['falcon', 'parrot', 'lion', 'monkey'],
...                   columns=('class', 'max_speed'))
>>> df
         class  max_speed
falcon    bird      389.0
parrot    bird       24.0
lion    mammal       80.5
monkey  mammal        NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
    class  max_speed
0    bird      389.0
1    bird       24.0
2  mammal       80.5
3  mammal        NaN
```

You can also use *reset_index* with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                    ('bird', 'parrot'),
...                                    ('mammal', 'lion'),
...                                    ('mammal', 'monkey')],
```

(continues on next page)

(continued from previous page)

```
...                                       names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                      ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
...                    ( 24.0, 'fly'),
...                    ( 80.5, 'run'),
...                    (np.nan, 'jump')],
...                   index=index,
...                   columns=columns)
>>> df
               speed species
                 max    type
class  name
bird   falcon  389.0     fly
       parrot   24.0     fly
mammal lion     80.5     run
       monkey    NaN    jump
```

If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
         class  speed species
                  max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
                speed species
         class    max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

When the index is inserted under another level, we can specify under which one with the parameter *col_fill*:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
              species  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

If we specify a nonexistent level for *col_fill*, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
                genus  speed species
                class    max    type
```

(continues on next page)

(continued from previous page)

```
name
falcon          bird  389.0     fly
parrot          bird   24.0     fly
lion          mammal   80.5     run
monkey        mammal    NaN    jump
```

rfloordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.floordiv

rmod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mod

rmul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mul

rolling(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculations.

New in version 0.18.0.

window [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. This is new in 0.19.0

min_periods [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

center [boolean, default False] Set the labels at the center of the window.

win_type [string, default None] Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

on [string, optional] For a DataFrame, column on which to calculate the rolling window, rather than the index

closed [string, default None] Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

axis : int or string, default 0

a Window or Rolling sub-classed for the particular operation

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
     B
0  NaN
1  1.0
```

(continues on next page)

(continued from previous page)

```
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                   index = [pd.Timestamp('20130101 09:00:00'),
...                            pd.Timestamp('20130101 09:00:02'),
...                            pd.Timestamp('20130101 09:00:03'),
...                            pd.Timestamp('20130101 09:00:05'),
...                            pd.Timestamp('20130101 09:00:06')])
```

```
>>> df
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```
>>> df.rolling('2s').sum()
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

To learn more about the offsets & frequency strings, please see this link.

The recognized win_types are:

- boxcar
- triang
- blackman
- hamming
- bartlett
- parzen
- bohman
- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general_gaussian (needs power, width)
- slepian (needs width).

If win_type=None all points are evenly weighted. To learn more about different window types see scipy.signal window functions.

expanding : Provides expanding transformations. ewm : Provides exponential weighted functions

round(*decimals=0*, **args*, ***kwargs*)

Round a DataFrame to a variable number of decimal places.

decimals [int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

```
>>> df = pd.DataFrame(np.random.random([3, 3]),
...     columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
>>> df
                A         B         C
first    0.028208  0.992815  0.173891
second   0.038683  0.645646  0.577595
third    0.877076  0.149370  0.491027
>>> df.round(2)
           A     B     C
first   0.03  0.99  0.17
second  0.04  0.65  0.58
third   0.88  0.15  0.49
>>> df.round({'A': 1, 'C': 2})
          A         B     C
first   0.0  0.992815  0.17
second  0.0  0.645646  0.58
third   0.9  0.149370  0.49
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
          A  B     C
first   0.0  1  0.17
```

(continues on next page)

(continued from previous page)

```
second  0.0  1  0.58
third   0.9  0  0.49
```

DataFrame object

numpy.around Series.round

rpow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.pow

rsub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
```

(continues on next page)

(continued from previous page)

```
...                    index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.sub

rtruediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*)

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

n [int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac [float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace [boolean, optional] Sample with or without replacement. Default = False.

weights [str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. inf and -inf values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

A new object of same type as caller.

Generate an example `Series` and `DataFrame`:

```
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0   -0.038497
1    1.820773
2   -0.972766
3   -1.598270
4   -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
          A         B         C         D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661
```

Next extract a random sample from both of these objects...

3 random elements from the `Series`:

```
>>> s.sample(n=3)
27   -0.994689
55   -1.049016
67   -0.224565
dtype: float64
```

And a random 10% of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.1, replace=True)
           A         B         C         D
35  1.981780  0.142106  1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640  0.217116
40  0.823173 -0.078816  1.009536  1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
6  -0.148339  0.832938  1.787600 -1.383767
```

You can use *random state* for reproducibility:

```
>>> df.sample(random_state=1)
A         B         C         D
37 -2.027662  0.103611  0.237496 -0.165867
43 -0.259323 -0.583426  1.516140 -0.479118
12 -1.686325 -0.579510  0.985195 -0.460286
8   1.167946  0.429082  1.215742 -1.636041
9   1.197475 -0.864188  1.554031 -1.505264
```

select (*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

crit [function] To be called on each index (label). Should return True or False

axis : int

selection : type of caller

select_dtypes(*include=None*, *exclude=None*)

Return a subset of the DataFrame's columns based on the column dtypes.

include, exclude [scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

ValueError

- If both of `include` and `exclude` are empty
- If `include` and `exclude` have overlapping elements
- If any kind of string dtype is passed in.

subset [DataFrame] The subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.

- To select all *numeric* types, use `np.number` or `'number'`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the numpy dtype hierarchy
- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`
- To select timedeltas, use `np.timedelta64`, `'timedelta'` or `'timedelta64'`
- To select Pandas categorical dtypes, use `'category'`
- To select Pandas datetimetz dtypes, use `'datetimetz'` (new in 0.20.0) or `'datetime64[ns, tz]'`

```
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
        a      b  c
0       1   True  1.0
1       2  False  2.0
2       1   True  1.0
3       2  False  2.0
4       1   True  1.0
5       2  False  2.0
```

```
>>> df.select_dtypes(include='bool')
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

```
>>> df.select_dtypes(include=['float64'])
   c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int'])
       b    c
0   True  1.0
1  False  2.0
2   True  1.0
3  False  2.0
4   True  1.0
5  False  2.0
```

sem(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

sem : Series or DataFrame (if level specified)

set_axis(*labels*, *axis=0*, *inplace=None*)
Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

labels [list-like, Index] The values for the new index.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

pandas.DataFrame.rename_axis : Alter the name of the index or columns.

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

set_index (*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

keys : column label or list of column labels / arrays drop : boolean, default True

Delete columns to be used as the new index

append [boolean, default False] Whether to append columns to existing index

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

verify_integrity [boolean, default False] Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale':[55, 40, 84, 31]})
   month  sale  year
0  1      55    2012
1  4      40    2014
2  7      84    2013
3  10     31    2014
```

Set the index to become the 'month' column:

```
>>> df.set_index('month')
       sale  year
month
1      55    2012
4      40    2014
7      84    2013
10     31    2014
```

Create a multi-index using columns 'year' and 'month':

```
>>> df.set_index(['year', 'month'])
            sale
year  month
2012  1     55
2014  4     40
2013  7     84
2014  10    31
```

Create a multi-index using a set of values and a column:

```
>>> df.set_index([[1, 2, 3, 4], 'year'])
         month  sale
   year
1  2012  1      55
2  2014  4      40
3  2013  7      84
4  2014  10     31
```

dataframe : DataFrame

set_value(*index*, *col*, *value*, *takeable=False*)

Put single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label value : scalar value takeable : interpret the index/col as indexers, default False

frame [DataFrame] If label pair is contained, will be reference to calling DataFrame, otherwise a new object

shape
Return a tuple representing the dimensionality of the DataFrame.

ndarray.shape

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

shift (*periods=1*, *freq=None*, *axis=0*)
Shift index by desired number of periods with an optional time freq

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, optional] Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

axis : {0 or 'index', 1 or 'columns'}

If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data.

shifted : DataFrame

size
Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

ndarray.size

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

skew (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

skew : Series or DataFrame (if level specified)

slice_shift (*periods=1*, *axis=0*)
Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

periods [int] Number of periods to move, can be positive or negative

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

shifted : same type as caller

sort_index (*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)
Sort object by labels (along an axis)

axis : index, columns to direct sorting level : int or level name or list of ints or list of level names

if not None, sort on values in specified index level(s)

ascending [boolean, default True] Sort ascending vs. descending

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining [bool, default True] if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

sorted_obj : DataFrame

sort_values (*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)
Sort by the values along either axis

by [str or list of str] Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels

Changed in version 0.23.0: Allow specifying index or column level names.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis to be sorted

ascending [bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end

sorted_obj : DataFrame

```
>>> df = pd.DataFrame({
...     'col1' : ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2' : [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
```

(continues on next page)

(continued from previous page)

```
... })
>>> df
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
3   NaN  8    4
4   D    7    2
5   C    4    3
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
    col1 col2 col3
1   A    1    1
0   A    2    0
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
    col1 col2 col3
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
3   NaN  8    4
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
    col1 col2 col3
3   NaN  8    4
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
```

sortlevel (*level=0*, *axis=0*, *ascending=True*, *inplace=False*, *sort_remaining=True*)

Sort multilevel index by chosen axis and primary level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order).

Deprecated since version 0.20.0: Use `DataFrame.sort_index()`

level : int axis : {0 or 'index', 1 or 'columns'}, default 0 ascending : boolean, default True inplace : boolean, default False

Sort the DataFrame without creating a new instance

sort_remaining [boolean, default True] Sort by the other levels too.

sorted : DataFrame

DataFrame.sort_index(level=...)

squeeze (*axis=None*)

Squeeze length 1 dimensions.

axis [None, integer or string axis name, optional] The axis to squeeze if 1-sized.

New in version 0.20.0.

scalar if 1-sized, else original object

stack (*level=-1*, *dropna=True*)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

The new index levels are sorted.

level [int, str, list, default -1] Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

dropna [bool, default True] Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of index and column values that are missing from the original dataframe. See Examples section.

DataFrame or Series Stacked dataframe or series.

DataFrame.unstack [Unstack prescribed level(s) from index axis] onto column axis.

DataFrame.pivot [Reshape dataframe from long format to wide] format.

DataFrame.pivot_table [Create a spreadsheet-style pivot table] as a DataFrame.

The function is named by analogy with a collection of books being re-organised from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of of each other (in the index of the dataframe).

Single level columns

```
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
     weight height
cat       0      1
dog       2      3
>>> df_single_level_cols.stack()
cat  weight    0
     height    1
dog  weight    2
     height    3
dtype: int64
```

Multi level columns: simple case

```
>>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol1)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
     weight
         kg    pounds
cat       1         2
dog       2         4
>>> df_multi_level_cols1.stack()
            weight
cat kg           1
    pounds       2
dog kg           2
    pounds       4
```

Missing values

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
    weight height
        kg      m
cat    1.0    2.0
dog    3.0    4.0
>>> df_multi_level_cols2.stack()
        height  weight
cat kg     NaN     1.0
    m      2.0     NaN
dog kg     NaN     3.0
    m      4.0     NaN
```

Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0)
             kg    m
cat height  NaN  2.0
    weight  1.0  NaN
dog height  NaN  4.0
    weight  3.0  NaN
>>> df_multi_level_cols2.stack([0, 1])
cat  height  m     2.0
     weight  kg    1.0
dog  height  m     4.0
     weight  kg    3.0
dtype: float64
```

Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
    weight height
        kg      m
cat    NaN    1.0
dog    2.0    3.0
>>> df_multi_level_cols3.stack(dropna=False)
        height  weight
cat kg     NaN     NaN
    m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
>>> df_multi_level_cols3.stack(dropna=True)
        height  weight
cat m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
```

std(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

std : Series or DataFrame (if level specified)

style
Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

pandas.io.formats.style.Styler

sub (*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

subtract (*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the sum of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

sum : Series or DataFrame (if level specified)

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

swapaxes (*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately

y : same as input

swaplevel (*i=-2*, *j=-1*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

swapped : type of caller (new object)

Changed in version 0.18.1: The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

tail (*n=5*)

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

n [int, default 5] Number of rows to select.

type of caller The last *n* rows of the caller object.

pandas.DataFrame.head : The first *n* rows of the caller object.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

take(*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

indices [array-like] An array of ints indicating which positions to take.

axis [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

taken [type of caller] An array-like containing the elements taken from the object.

DataFrame.loc : Select a subset of a DataFrame by labels. DataFrame.iloc : Select a subset of a DataFrame by positions. numpy.take : Take elements from an array along an axis.

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                    columns=['name', 'class', 'max_speed'],
...                    index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

to_clipboard(*excel=True*, *sep=None*, ***kwargs*)

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

excel [bool, default True]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

sep [str, default `'\t'`] Field delimiter.

****kwargs** These parameters will be passed to DataFrame.to_csv.

DataFrame.to_csv [Write a DataFrame to a comma-separated values] (csv) file.

read_clipboard : Read text from clipboard and pass to read_table.

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

to_csv(*path_or_buf=None*, *sep=', '*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *encoding=None*, *compression=None*, *quoting=None*, *quotechar='"'*, *line_terminator='\n'*, *chunksize=None*, *tupleize_cols=None*, *date_format=None*, *doublequote=True*, *escapechar=None*, *decimal='.'*)

Write DataFrame to a comma-separated values (csv) file

path_or_buf [string or file handle, default None] File path or object, if None is provided the result is returned as a string.

sep [character, default ','] Field delimiter for the output file.

na_rep [string, default ''] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R

mode [str] Python write mode, default 'w'

encoding [string, optional] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

compression [string, optional] A string representing the compression to use in the output file. Allowed values are 'gzip', 'bz2', 'zip', 'xz'. This input is only used when the first argument is a filename.

line_terminator [string, default `'\n'`] The newline character or character sequence to use in the output file

quoting [optional constant from csv module] defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric

quotechar [string (length 1), default '"'] character used to quote fields

doublequote [boolean, default True] Control quoting of *quotechar* inside a field

escapechar [string (length 1), default None] character used to escape *sep* and *quotechar* when appropriate

chunksize [int or None] rows to write at a time

tupleize_cols [boolean, default False] Deprecated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

Write MultiIndex columns as a list of tuples (if True) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if False).

date_format [string, default None] Format string for datetime objects

decimal: string, default '.' Character recognized as decimal separator. E.g. use ',' for European data

to_dense()
Return dense representation of NDFrame (as opposed to sparse)

to_dict(*orient='dict'*, *into=<type 'dict'>*)
Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

orient [str {'dict', 'list', 'series', 'split', 'records', 'index'}] Determines the type of the values of the dictionary.

- 'dict' (default) : dict like {column -> {index -> value}}
- 'list' : dict like {column -> [values]}
- 'series' : dict like {column -> Series(values)}
- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'records' : list like [{column -> value}, ... , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

into [class, default dict] The collections.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

New in version 0.21.0.

result : collections.Mapping like {column -> {index -> value}}

DataFrame.from_dict: create a DataFrame from a dictionary DataFrame.to_json: convert a DataFrame to JSON format

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   col1  col2
a     1   0.50
b     2   0.75
>>> df.to_dict()
{'col1': {'a': 1, 'b': 2}, 'col2': {'a': 0.5, 'b': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': a    1
         b    2
         Name: col1, dtype: int64,
 'col2': a    0.50
         b    0.75
         Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['a', 'b'], 'columns': ['col1', 'col2'],
 'data': [[1.0, 0.5], [2.0, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1.0, 'col2': 0.5}, {'col1': 2.0, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'a': {'col1': 1.0, 'col2': 0.5}, 'b': {'col1': 2.0, 'col2': 0.75}}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('a', 1), ('b', 2)])),
             ('col2', OrderedDict([('a', 0.5), ('b', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1.0, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2.0, 'col2': 0.75})]
```

to_excel(*excel_writer*, *sheet_name='Sheet1'*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *startrow=0*, *startcol=0*, *engine=None*, *merge_cells=True*, *encoding=None*, *inf_rep='inf'*, *verbose=True*, *freeze_panes=None*)

Write DataFrame to an excel sheet

excel_writer [string or ExcelWriter object] File path or existing ExcelWriter

sheet_name [string, default 'Sheet1'] Name of sheet which will contain DataFrame

na_rep [string, default ''] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow : upper left cell row to dump data frame

startcol : upper left cell column to dump data frame

engine [string, default None] write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

merge_cells [boolean, default True] Write MultiIndex and Hierarchical Rows as merged cells.

encoding: string, default None encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep [string, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel)

freeze_panes [tuple of integer (length 2), default None] Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

For compatibility with to_csv, to_excel serializes lists and dicts to strings before writing.

to_feather(*fname*)
write out the binary feather-format for DataFrames

New in version 0.20.0.

fname [str] string file path

to_gbq(*destination_table*, *project_id*, *chunksize=None*, *verbose=None*, *reauth=False*, *if_exists='fail'*, *private_key=None*, *auth_local_webserver=False*, *table_schema=None*)
Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

Authentication to the Google BigQuery service is via OAuth 2.0.

- If `private_key` is provided, the library loads the JSON service account credentials and uses those to authenticate.
- If no `private_key` is provided, the library tries application default credentials.
- If application default credentials are not found or cannot be used with BigQuery, the library authenticates with user account credentials. In this case, you will be asked to grant permissions for product name 'pandas GBQ'.

destination_table [str] Name of table to be written, in the form 'dataset.tablename'.

project_id [str] Google BigQuery Account project ID.

chunksize [int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

reauth [bool, default False] Force Google BigQuery to reauthenticate the user. This is useful if multiple accounts are used.

if_exists [str, default 'fail'] Behavior when the destination table exists. Value can be one of:

'fail' If table exists, do nothing.

'replace' If table exists, drop it, recreate it, and insert data.

'append' If table exists, insert data. Create if does not exist.

private_key [str, optional] Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

auth_local_webserver [bool, default False] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

table_schema [list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'},...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

New in version 0.3.1 of pandas-gbq.

verbose [boolean, deprecated] *Deprecated in Pandas-GBQ 0.4.0.* Use the logging module to adjust verbosity instead.

pandas_gbq.to_gbq : This function in the pandas-gbq library. pandas.read_gbq : Read a DataFrame from Google BigQuery.

to_hdf (*path_or_buf*, *key*, ***kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the user guide.

path_or_buf [str or pandas.HDFStore] File path or HDFStore object.

key [str] Identifier for the group in the store.

mode [{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

format [{'fixed', 'table'}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

append [bool, default False] For Table formats, append the input data to the existing.

data_columns [list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See io.hdf5-query-data-columns. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

errors [str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

DataFrame.read_hdf : Read from HDF file. DataFrame.to_parquet : Write a DataFrame to the binary parquet format. DataFrame.to_sql : Write to a sql table. DataFrame.to_feather : Write out feather-format for DataFrames. DataFrame.to_csv : Write out to a csv file.

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

to_html (*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *bold_rows=True*, *classes=None*, *escape=True*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*, *notebook=False*, *decimal='.'*, *border=None*, *table_id=None*)

Render a DataFrame as an HTML table.

to_html-specific options:

bold_rows [boolean, default True] Make the row labels bold in the output

classes [str or list or tuple, default None] CSS class(es) to apply to the resulting html table

escape [boolean, default True] Convert the characters <, >, and & to HTML-safe sequences.

max_rows [int, optional] Maximum number of rows to show before truncating. If None, show all.

max_cols [int, optional] Maximum number of columns to show before truncating. If None, show all.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe

New in version 0.18.0.

border [int] A `border=border` attribute is included in the opening <*table*> tag. Default `pd.options.html.border`.

New in version 0.19.0.

table_id [str, optional] A css id is included in the opening <*table*> tag if specified.

New in version 0.23.0.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] whether to print column labels, default True

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_json(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression=None*, *index=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

path_or_buf [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'

  - allowed values are: {‘split’,’records’,’index’}
- DataFrame
  - default is ‘columns’
  - allowed values are: {‘split’,’records’,’index’,’columns’,’values’}
- The format of the JSON string
  - ‘split’ : dict like {‘index’ -> [index], ‘columns’ -> [columns], ‘data’ -> [values]}
  - ‘records’ : list like [{column -> value}, … , {column -> value}]
  - ‘index’ : dict like {index -> {column -> value}}
  - ‘columns’ : dict like {column -> {index -> value}}
  - ‘values’ : just the values array
  - ‘table’ : dict like {‘schema’: {schema}, ‘data’: {data}} describing the data, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

date_format [{None, ‘epoch’, ‘iso’}] Type of date conversion. ‘epoch’ = epoch milliseconds, ‘iso’ = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is ‘iso’. For all other orients, the default is ‘epoch’.

double_precision [int, default 10] The number of decimal places to use when encoding floating point values.

force_ascii [boolean, default True] Force encoded string to be ASCII.

date_unit [string, default ‘ms’ (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’, ‘ns’ for second, millisecond, microsecond, and nanosecond respectively.

default_handler [callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines [boolean, default False] If ‘orient’ is ‘records’ write out line delimited json format. Will throw ValueError if incorrect ‘orient’ since others are not list like.

New in version 0.19.0.

compression [{None, ‘gzip’, ‘bz2’, ‘zip’, ‘xz’}] A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

index [boolean, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is ‘split’ or ‘table’.

New in version 0.23.0.

pandas.read_json

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]}'
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

to_latex(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=False*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires \usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

to_latex-specific options:

bold_rows [boolean, default False] Make the row labels bold in the output

column_format [str, default None] The columns format as specified in LaTeX table format e.g 'rcl' for 3 columns

longtable [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a \usepackage{longtable} to your LaTeX preamble.

escape [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

encoding [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

multicolumn [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module.

New in version 0.20.0.

multirow [boolean, default False] Use multirow to enhance MultiIndex rows. Requires adding a \usepackage{multirow} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

New in version 0.20.0.

to_msgpack (*path_or_buf=None*, *encoding='utf-8'*, ***kwargs*)
msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

path [string File path, buffer-like, or None] if None, return generated string

append [boolean whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

to_panel ()
Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Deprecated since version 0.20.0.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

panel : Panel

to_parquet (*fname*, *engine='auto'*, *compression='snappy'*, ***kwargs*)
Write a DataFrame to the binary parquet format.

New in version 0.21.0.

This function writes the dataframe as a parquet file. You can choose different parquet backends, and have the option of compression. See the user guide for more details.

fname [str] String file path.

engine [{'auto', 'pyarrow', 'fastparquet'}, default 'auto'] Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

compression [{'snappy', 'gzip', 'brotli', None}, default 'snappy'] Name of the compression to use. Use `None` for no compression.

****kwargs** Additional arguments passed to the parquet library. See pandas io for more details.

read_parquet : Read a parquet file. DataFrame.to_csv : Write a csv file. DataFrame.to_sql : Write to a sql table. DataFrame.to_hdf : Write to hdf.

This function requires either the fastparquet or pyarrow library.

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gzip', compression='gzip')
>>> pd.read_parquet('df.parquet.gzip')
   col1  col2
```

(continues on next page)

(continued from previous page)

```
0      1      3
1      2      4
```

to_period (*freq=None*, *axis=0*, *copy=True*)

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

freq : string, default axis : {0 or 'index', 1 or 'columns'}, default 0

The axis to convert (the index by default)

copy [boolean, default True] If False then underlying input data is not copied

ts : TimeSeries with PeriodIndex

to_pickle (*path*, *compression='infer'*, *protocol=2*)

Pickle (serialize) object to file.

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

protocol [int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1]_ paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

read_pickle : Load pickled pandas object (or any object) from file. DataFrame.to_hdf : Write DataFrame to an HDF5 file. DataFrame.to_sql : Write DataFrame to a SQL database. DataFrame.to_parquet : Write a DataFrame to the binary parquet format.

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

to_records(*index=True*, *convert_datetime64=None*)
Convert DataFrame to a NumPy record array.

Index will be put in the 'index' field of the record array if requested.

index [boolean, default True] Include index in resulting record array, stored in 'index' field.

convert_datetime64 [boolean, default None] Deprecated since version 0.23.0.

Whether to convert the index to datetime.datetime if it is a DatetimeIndex.

y : numpy.recarray

DataFrame.from_records: convert structured or record ndarray to DataFrame.

numpy.recarray: ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   A     B
a  1  0.50
b  2  0.75
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

By default, timestamps are converted to *datetime.datetime*:

```
>>> df.index = pd.date_range('2018-01-01 09:00', periods=2, freq='min')
>>> df
                     A     B
2018-01-01 09:00:00  1  0.50
2018-01-01 09:01:00  2  0.75
>>> df.to_records()
rec.array([(datetime.datetime(2018, 1, 1, 9, 0), 1, 0.5 ),
           (datetime.datetime(2018, 1, 1, 9, 1), 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The timestamp conversion can be disabled so NumPy's datetime64 data type is used instead:

```
>>> df.to_records(convert_datetime64=False)
rec.array([('2018-01-01T09:00:00.000000000', 1, 0.5 ),
           ('2018-01-01T09:01:00.000000000', 2, 0.75)],
          dtype=[('index', '<M8[ns]'), ('A', '<i8'), ('B', '<f8')])
```

to_sparse(*fill_value=None*, *kind='block'*)
Convert to SparseDataFrame

fill_value : float, default NaN kind : {'block', 'integer'}

y : SparseDataFrame

to_sql(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1]_ are supported. Tables can be newly created, appended to, or overwritten.

name [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists [{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index [boolean, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize [int, optional] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

ValueError When the table already exists and *if_exists* is 'fail' (the default).

pandas.read_sql : read a DataFrame from a table

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
```

(continues on next page)

(continued from previous page)

```
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

to_stata(*fname*, *convert_dates=None*, *write_index=True*, *encoding='latin-1'*, *byteorder=None*, *time_stamp=None*, *data_label=None*, *variable_labels=None*, *version=114*, *convert_strl=None*)

Export Stata binary dta files.

fname [path (string), buffer or path object] string, path object (pathlib.Path or py._path.local.LocalPath) or object implementing a binary write() functions. If using a buffer then the buffer will not be automatically closed after the file data has been written.

convert_dates [dict] Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to 'tc'. Raises NotImplementedError if a datetime column has timezone information.

write_index [bool] Write the index to Stata dataset.

encoding [str] Default is latin-1. Unicode is not supported.

byteorder [str] Can be ">", "<", "little", or "big". default is *sys.byteorder*.

time_stamp [datetime] A datetime to use as file creation date. Default is the current time.

data_label [str] A label for the data set. Must be 80 characters or smaller.

variable_labels [dict] Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

New in version 0.19.0.

version [{114, 117}] Version to use in the output dta file. Version 114 can be used read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 114 limits string variables to 244 characters

or fewer while 117 allows strings with lengths up to 2,000,000 characters.

New in version 0.23.0.

convert_strl [list, optional] List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

New in version 0.23.0.

NotImplementedError

- If datetimes contain timezone information
- Column dtype is not representable in Stata

ValueError

- Columns listed in convert_dates are neither datetime64[ns] or datetime.datetime
- Column listed in convert_dates is not in DataFrame
- Categorical label contains more than 32,000 characters

New in version 0.19.0.

pandas.read_stata : Import Stata data files pandas.io.stata.StataWriter : low-level writer for Stata data files pandas.io.stata.StataWriter117 : low-level writer for version 117 files

```
>>> data.to_stata('./data_file.dta')
```

Or with dates

```
>>> data.to_stata('./date_data_file.dta', {2 : 'tw'})
```

Alternatively you can create an instance of the StataWriter class

```
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

With dates:

```
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
```

to_string(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *line_width=None*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*)

Render a DataFrame to a console-friendly tabular output.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_timestamp (*freq=None*, *how='start'*, *axis=0*, *copy=True*)
Cast to DatetimeIndex of timestamps, at *beginning* of period

freq [string, default frequency of PeriodIndex] Desired frequency

how [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default)

copy [boolean, default True] If false then underlying input data is not copied

df : DataFrame with DatetimeIndex

to_xarray()
Return an xarray object from the pandas object.

a DataArray for a Series a Dataset for a DataFrame a DataArray for higher dims

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
```

(continues on next page)

(continued from previous page)

```
   A    B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index    (index) int64 0 1 2
Data variables:
    A        (index) int64 1 1 2
    B        (index) object 'foo' 'bar' 'foo'
    C        (index) float64 4.0 5.0 6.0
```

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)}
                      ).set_index(['B','A'])
>>> df
         C
B   A
foo 1  4.0
bar 1  5.0
foo 2  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B        (B) object 'bar' 'foo'
  * A        (A) int64 1 2
Data variables:
    C        (B, A) float64 5.0 nan 4.0 6.0
```

```
>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                 items=list('ABCD'),
                 major_axis=pd.date_range('20130101', periods=3),
                 minor_axis=['first', 'second'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second
```

```
>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],
       [[ 6,  7],
        [ 8,  9],
        [10, 11]],
       [[12, 13],
```

(continues on next page)

(continued from previous page)

```
        [14, 15],
        [16, 17]],
       [[18, 19],
        [20, 21],
        [22, 23]]])
Coordinates:
  * items       (items) object 'A' 'B' 'C' 'D'
  * major_axis  (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03
↪# noqa
  * minor_axis  (minor_axis) object 'first' 'second'
```

See the xarray docs

transform(*func*, **args*, ***kwargs*)

Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values

New in version 0.20.0.

func [callable, string, dictionary, or list of string/callables] To apply to column

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

transformed : NDFrame

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
...                   index=pd.date_range('1/1/2000', periods=10))
df.iloc[3:7] = np.nan
```

```
>>> df.transform(lambda x: (x - x.mean()) / x.std())
                   A         B         C
2000-01-01  0.579457  1.236184  0.123424
2000-01-02  0.370357 -0.605875 -1.231325
2000-01-03  1.455756 -0.277446  0.288967
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -0.498658  1.274522  1.642524
2000-01-09 -0.540524 -1.012676 -0.828968
2000-01-10 -1.366388 -0.614710  0.005378
```

pandas.NDFrame.aggregate pandas.NDFrame.apply

transpose(**args*, ***kwargs*)

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
                0     1
name        Alice   Bob
score         9.5     8
employed    False  True
kids            0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

truediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

before [date, string, int] Truncate all rows before this index value.

after [date, string, int] Truncate all rows after this index value.

axis [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

copy [boolean, default is True,] Return a copy of the truncated section.

type of caller The truncated Series or DataFrame.

DataFrame.loc : Select a subset of a DataFrame by label. DataFrame.iloc : Select a subset of a DataFrame by position.

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
```

(continues on next page)

(continued from previous page)

```
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

tshift (*periods=1*, *freq=None*, *axis=0*)
Shift the time index, using the index's frequency if available.

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

shifted : NDFrame

tz_convert (*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

tz : string or pytz.timezone object axis : the axis to convert level : int, str, default None

If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

TypeError If the axis is tz-naive.

tz_localize (*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*)
Localize tz-naive TimeSeries to target time zone.

tz : string or pytz.timezone object axis : the axis to localize level : int, str, default None

If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise']

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times

- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

TypeError If the TimeSeries is tz-aware and tz is not None.

unstack (*level=-1*, *fill_value=None*)

Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels. If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex). The level involved will automatically get sorted.

level [int, string, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name

fill_value [replace NaN with this value if the unstack produces] missing values

New in version 0.18.0.

DataFrame.pivot : Pivot a table based on column values. DataFrame.stack : Pivot a level of the column labels (inverse operation

from *unstack*).

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                    ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a   1.0
     b   2.0
two  a   3.0
     b   4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a   b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
   one  two
a  1.0   3.0
b  2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a  1.0
     b  2.0
two  a  3.0
     b  4.0
dtype: float64
```

unstacked : DataFrame or Series

update (*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

other [DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

join [{'left'}, default 'left'] Only left join is implemented, keeping the index and columns of the original object.

overwrite [bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.
- False: only update values that are NA in the original DataFrame.

filter_func [callable(1d-array) -> boolean 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

raise_conflict [bool, default False] If True, will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

ValueError When *raise_conflict* is True and there's overlapping non-NA data.

dict.update : Similar method for dictionaries. DataFrame.merge : For column(s)-on-columns(s) operations.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
1  b  e
2  c  f
```

For Series, it's name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
```

(continues on next page)

(continued from previous page)

```
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A      B
0  1    4.0
1  2  500.0
2  3    6.0
```

values

Return a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

numpy.ndarray The values of the DataFrame.

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age':    [ 3,  29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3      94      31
1   29     170     115
>>> df.dtypes
age       int64
height    int64
weight    int64
dtype: object
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('monkey', np.nan, None)],
...                    columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name          object
max_speed    float64
rank          object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
```

(continues on next page)

(continued from previous page)

```
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

pandas.DataFrame.index : Retrievie the index labels pandas.DataFrame.columns : Retrieving the column names

var(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

var : Series or DataFrame (if level specified)

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
```

(continues on next page)

(continued from previous page)

```
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

DataFrame.mask()

xs(*key*, *axis=0*, *level=None*, *drop_level=True*)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on the rows (axis=0).

key [object] Some label contained in the index, or partially in a MultiIndex

axis [int, default 0] Axis to retrieve cross-section on

level [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level [boolean, default True] If False, returns object with same levels as self.

```
>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
A    4
B    5
C    2
Name: a
>>> df.xs('C', axis=1)
a    2
b    9
c    3
Name: C
```

```
>>> df
                     A  B  C  D
first second third
bar   one    1       4  1  8  9
      two    1       7  5  5  0
baz   one    1       6  6  8  0
      three  2       5  3  5  3
>>> df.xs(('baz', 'three'))
       A  B  C  D
third
2      5  3  5  3
>>> df.xs('one', level=1)
             A  B  C  D
first third
bar   1      4  1  8  9
baz   1      6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
        A  B  C  D
```

(continues on next page)

(continued from previous page)

```
second
three   5  3  5  3
```

xs : Series or DataFrame

xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of xs functionality, see MultiIndex Slicers

class Fred2.Core.Result.**Distance2SelfResult**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

Bases: *Fred2.Core.Result.AResult*

Distance2Self prediction result

T

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
```

(continues on next page)

(continued from previous page)

```
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
              0     1
name      Alice   Bob
score       9.5     8
employed  False  True
kids          0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name          object
score        float64
employed        bool
kids           int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

abs()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

abs Series/DataFrame containing the absolute value of each element.

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
     a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
     a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

numpy.absolute : calculate the absolute value element-wise.

add(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
```

(continues on next page)

(continued from previous page)

```
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.radd

add_prefix(*prefix*)

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

prefix [str] The string to add before each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_suffix: Suffix row labels with string *suffix*. DataFrame.add_suffix: Suffix column labels with string *suffix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0       1       3
1       2       4
2       3       5
3       4       6
```

add_suffix(*suffix*)

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

suffix [str] The string to add after each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_prefix: Prefix row labels with string *prefix*. DataFrame.add_prefix: Prefix column labels with string *prefix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

agg(*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

aggregate(*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

align(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)
Align two objects on their axes with the specified join method for each axis Index

other : DataFrame or Series join : {'outer', 'inner', 'left', 'right'}, default 'outer' axis : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None)

level [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level

copy [boolean, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

method : str, default None limit : int, default None fill_axis : {0 or 'index', 1 or 'columns'}, default 0

Filling axis, method and limit

broadcast_axis [{0 or 'index', 1 or 'columns'}, default None] Broadcast values along this axis, if aligning two objects of different dimensions

(left, right) [(DataFrame, type of other)] Aligned objects

all(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether all elements are True, potentially over an axis.

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.

- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

all : Series or DataFrame (if level specified)

pandas.Series.all : Return True if all elements are True pandas.DataFrame.any : Return True if one (or more) elements are True

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

any (*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

any : Series or DataFrame (if level specified)

pandas.DataFrame.all : Return whether all elements are True.

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([True, False]).any()
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
```

(continues on next page)

(continued from previous page)

```
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

append(*other*, *ignore_index=False*, *verify_integrity=False*, *sort=None*)

Append rows of *other* to the end of this frame, returning a new object. Columns not in this frame are added as new columns.

other [DataFrame or Series/dict-like object, or list of these] The data to append.

ignore_index [boolean, default False] If True, do not use the index labels.

verify_integrity [boolean, default False] If True, raise ValueError on creating index with duplicates.

sort [boolean, default None] Sort columns if the columns of *self* and *other* are not aligned. The default sorting is deprecated and will change to not-sorting in a future version of pandas. Explicitly pass `sort=True` to silence the warning and sort. Explicitly pass `sort=False` to silence the warning and not sort.

New in version 0.23.0.

appended : DataFrame

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

pandas.concat [General function to concatenate DataFrame, Series] or Panel objects

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With *ignore_index* set to True:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
...     df = df.append({'A': i}, ignore_index=True)
>>> df
   A
0  0
1  1
2  2
3  3
4  4
```

More efficient:

```
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
...           ignore_index=True)
   A
0  0
1  1
2  2
3  3
4  4
```

apply (*func*, *axis=0*, *broadcast=None*, *raw=False*, *reduce=None*, *result_type=None*, *args=()*, ***kwds*)
Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

func [function] Function to apply to each column or row.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

broadcast [bool, optional] Only relevant for aggregation functions:

- `False` or `None` : returns a Series whose length is the length of the index or the number of columns (based on the *axis* parameter)
- `True` : results will be broadcast to the original shape of the frame, the original index and columns will be retained.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by result_type='broadcast'.

raw [bool, default False]

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

reduce [bool or None, default None] Try to apply reduction procedures. If the DataFrame is empty, *apply* will use *reduce* to determine whether the result should be a Series or a DataFrame. If `reduce=None` (the default), *apply*'s return value will be guessed by calling *func* on an empty Series (note: while guessing, exceptions raised by *func* will be ignored). If `reduce=True` a Series will always be returned, and if `reduce=False` a DataFrame will always be returned.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by `result_type='reduce'`.

result_type [{'expand', 'reduce', 'broadcast', None}, default None] These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.
- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

New in version 0.23.0.

args [tuple] Positional arguments to pass to *func* in addition to the array/series.

****kwds** Additional keyword arguments to pass as keywords arguments to *func*.

In the current implementation apply calls *func* twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if *func* has side-effects, as they will take effect twice for the first column/row.

DataFrame.applymap: For elementwise operations DataFrame.aggregate: only perform aggregating type operations DataFrame.transform: only perform transformating type operations

```
>>> df = pd.DataFrame([[4, 9],] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Retuning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing result_type='expand' will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
1  1  2
2  1  2
```

applied : Series or DataFrame

applymap (*func*)

Apply a function to a Dataframe elementwise.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

func [callable] Python function, returns a single value from a single value.

DataFrame Transformed DataFrame.

DataFrame.apply : Apply a function along input axis of DataFrame

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.applymap(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.applymap(lambda x: x**2)
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

But it's better to avoid applymap in that case.

```
>>> df ** 2
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

as_blocks(*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

copy : boolean, default True

values : a dict of dtype -> Constructor Types

as_matrix(*columns=None*)

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

columns: list, optional, default:None If None, return all columns, otherwise, returns specified columns.

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a flot64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

pandas.DataFrame.values

asfreq(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)

Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

freq : DateOffset object, or string method : {'backfill'/'bfill', 'pad'/'ffill'}, default None

Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill

how [{'start', 'end'}, default end] For PeriodIndex only, see PeriodIndex.asfreq

normalize [bool, default False] Whether to reset output index to midnight

fill_value: scalar, optional Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

converted : type of caller

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:03:00    3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  NaN
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  2.0
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  3.0
2000-01-01 00:03:00  3.0
```

reindex

To learn more about the frequency strings, please see this link.

asof(*where*, *subset=None*)

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

where : date or array of dates subset : string or list of strings, default None

if not None use these columns for NaN propagation

Dates are assumed to be sorted Raises if this is not the case

where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

where is Index: same shape object as input

merge_asof

assign(***kwargs*)

Assign new columns to a DataFrame, returning a new object (a copy) with the new columns added to the original ones. Existing columns that are re-assigned will be overwritten.

kwargs [keyword, value pairs] keywords are the column names. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

df [DataFrame] A new DataFrame with the new columns in addition to all the existing columns.

Assigning multiple columns within the same `assign` is possible. For Python 3.6 and above, later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order. For Python 3.5 and below, the order of keyword arguments is not specified, you cannot refer to newly created or modified columns. All items are computed first, and then assigned in alphabetical order.

Changed in version 0.23.0: Keyword argument order is maintained for Python 3.6 and later.

```
>>> df = pd.DataFrame({'A': range(1, 11), 'B': np.random.randn(10)})
```

Where the value is a callable, evaluated on *df*:

```
>>> df.assign(ln_A = lambda x: np.log(x.A))
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the value already exists and is inserted:

```
>>> newcol = np.log(df['A'])
>>> df.assign(ln_A=newcol)
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the keyword arguments depend on each other

```
>>> df = pd.DataFrame({'A': [1, 2, 3]})
```

```
>>> df.assign(B=df.A, C=lambda x:x['A']+ x['B'])
   A  B  C
0  1  1  2
1  2  2  4
2  3  3  6
```

astype(***kwargs*)

Cast a pandas object to a specified dtype `dtype`.

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy [bool, default True.] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors [{'raise', 'ignore'}, default 'raise'.] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised

- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

raise_on_error [raise on invalid input] Deprecated since version 0.20.0: Use `errors` instead

kwargs : keyword arguments to pass on to the constructor

casted : type of caller

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> ser.astype('category', ordered=True, categories=[2, 1])
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to a numeric type. numpy.ndarray.astype : Cast a numpy array to a specified type.

at

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.iat [Access a single value for a row/column pair by integer] position

DataFrame.loc : Access a group of rows and columns by label(s) Series.at : Access a single value using a label

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

KeyError When label does not exist in DataFrame

at_time (*time*, *asof=False*)

Select values at particular time of day (e.g. 9:30AM).

TypeError If the index is not a `DatetimeIndex`

time : datetime.time or string

values_at_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

between_time : Select values between particular times of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_at_time : Get just the index locations for

values at particular time of the day

axes

Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['col1', 'col2'],
dtype='object')]
```

between_time (*start_time*, *end_time*, *include_start=True*, *include_end=True*)
Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

TypeError If the index is not a `DatetimeIndex`

start_time : datetime.time or string end_time : datetime.time or string include_start : boolean, default True include_end : boolean, default True

values_between_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

at_time : Select values at a particular time of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_between_time : Get just the index locations for

values between particular times of the day

bfill (*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='bfill')`

blocks
Internal property, property synonym for as_blocks()

Deprecated since version 0.21.0.

bool ()
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

boxplot(*column=None*, *by=None*, *ax=None*, *fontsize=None*, *rot=0*, *grid=True*, *figsize=None*, *layout=None*, *return_type=None*, ***kwds*)

Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to *1.5 * IQR (IQR = Q3 - Q1)* from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia's entry for boxplot.

column [str or list of str, optional] Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.

by [str or array-like, optional] Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in *by*.

ax [object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.

fontsize [float or str] Tick label font size in points or as a string (e.g., *large*).

rot [int or float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate sytem.

grid [boolean, default True] Setting this to True will show the grid.

figsize [A tuple (width, height) in inches] The size of the figure to create in matplotlib.

layout [tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 columns and 5 rows, starting from the top-left.

return_type [{'axes', 'dict', 'both'} or None, default 'axes'] The kind of object to return. The default is `axes`.

- 'axes' returns the matplotlib axes the boxplot is drawn on.
- 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot.
- 'both' returns a namedtuple with the axes and dict.
- when grouping with `by`, a Series mapping columns to `return_type` is returned.

  If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

result :

The return type depends on the *return_type* parameter:

- 'axes' : object of class matplotlib.axes.Axes
- 'dict' : dict of matplotlib.lines.Line2D objects
- 'both' : a nametuple with strucure (ax, lines)

For data grouped with `by`:

- `Series`
- `array` (for `return_type = None`)

Series.plot.hist: Make a histogram. matplotlib.pyplot.boxplot : Matplotlib equivalent plot.

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`. For instance:

A list of strings (i.e. `['X', 'Y']`) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

The layout of boxplot can be adjusted giving a tuple to `layout`:

Additional formatting can be done to the boxplot, like suppressing the grid (`grid=False`), rotating the labels in the x-axis (i.e. `rot=45`) or changing the fontsize (i.e. `fontsize=15`):

The parameter `return_type` can be used to select the type of element returned by *boxplot*. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```
>>> boxplot = df.boxplot(column=['Col1','Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

When grouping with `by`, a Series mapping columns to `return_type` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned:

```
>>> boxplot =  df.boxplot(column=['Col1', 'Col2'], by='X',
...                       return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

clip (*lower=None*, *upper=None*, *axis=None*, *inplace=False*, **args*, ***kwargs*)
Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

lower [float or array_like, default None] Minimum threshold value. All values below this threshold will be set to it.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

clip_lower : Clip values below specified threshold(s). clip_upper : Clip values above specified threshold(s).

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

clip_lower(*threshold*, *axis=None*, *inplace=False*)

Return copy of the input with values below a threshold truncated.

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Series.clip [Return copy of input with values below and above] thresholds truncated.

Series.clip_upper [Return copy of input with values above] threshold truncated.

clipped : same type as input

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

clip_upper (*threshold*, *axis=None*, *inplace=False*)
Return copy of input with values above given value(s) truncated.

threshold : float or array_like axis : int or string axis name, optional

Align object with threshold along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data

New in version 0.21.0.

clip

clipped : same type as input

columns
The column labels of the DataFrame.

combine (*other*, *func*, *fill_value=None*, *overwrite=True*)
Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

other : DataFrame func : function

Function that takes two series as inputs and return a Series or a scalar

fill_value : scalar value overwrite : boolean, default True

If True then overwrite values for common keys in the calling frame

result : DataFrame

```
>>> df1 = DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, lambda s1, s2: s1 if s1.sum() < s2.sum() else s2)
   A  B
0  0  3
1  0  3
```

DataFrame.combine_first [Combine two DataFrame objects and default to] non-null values in frame calling the method

combine_first (*other*)
Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

other : DataFrame

combined : DataFrame

df1's values prioritized, use values from df2 to fill holes:

```
>>> df1 = pd.DataFrame([[1, np.nan]])
>>> df2 = pd.DataFrame([[3, 4]])
>>> df1.combine_first(df2)
   0    1
0  1  4.0
```

DataFrame.combine [Perform series-wise operation on two DataFrames] using a given function

compound(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

compounded : Series or DataFrame (if level specified)

consolidate(*inplace=False*)

Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

convert_objects(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Return a fixed frequency timedelta index,

with day as the default.

converted : same as input object

copy(*deep=True*)

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

deep [bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

copy [Series, DataFrame or Panel] Object type matches caller.

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
>>> s
a    3
b    4
dtype: int64
```

(continues on next page)

(continued from previous page)

```
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

corr(*method='pearson'*, *min_periods=1*)

Compute pairwise correlation of columns, excluding NA/null values

method [{'pearson', 'kendall', 'spearman'}]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation

y : DataFrame

corrwith(*other*, *axis=0*, *drop=False*)

Compute pairwise correlation between rows or columns of two DataFrame objects.

other : DataFrame, Series axis : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' to compute column-wise, 1 or 'columns' for row-wise

drop [boolean, default False] Drop missing indices from result, default returns union of all

correls : Series

count(*axis=0*, *level=None*, *numeric_only=False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, and optionally *numpy.inf* (depending on *pandas.options.mode.use_inf_as_na*) are considered NA.

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each **row**.

level [int or str, optional] If the axis is a *MultiIndex* (hierarchical), count along a particular *level*, collapsing into a *DataFrame*. A *str* specifies the level name.

numeric_only [boolean, default False] Include only *float*, *int* or *boolean* data.

Series or DataFrame For each column/row the number of non-NA/null entries. If *level* is specified returns a *DataFrame*.

Series.count: number of non-NA elements in a Series DataFrame.shape: number of DataFrame rows and columns (including NA

elements)

DataFrame.isna: boolean same-sized DataFrame showing places of NA elements

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                    ["John", "Myla", None, "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> df
   Person   Age  Single
0    John  24.0   False
1    Myla   NaN    True
2    None  21.0    True
3    John  33.0    True
4    Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    4
Age       4
Single    5
dtype: int64
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    2
3    3
4    3
dtype: int64
```

Counts for one level of a *MultiIndex*:

```
>>> df.set_index(["Person", "Single"]).count(level="Person")
        Age
Person
John      2
Myla      1
```

cov(*min_periods=None*)

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as `NaN`.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result.

DataFrame The covariance matrix of the series of the DataFrame.

pandas.Series.cov : compute covariance with another Series pandas.core.window.EWM.cov: expoential weighted sample covariance pandas.core.window.Expanding.cov : expanding sample covariance pandas.core.window.Rolling.cov : rolling sample covariance

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                   columns=['dogs', 'cats'])
>>> df.cov()
          dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
          a         b         c         d         e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
```

(continues on next page)

(continued from previous page)

```
          a         b         c
a  0.316741       NaN -0.150812
b       NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

cummax (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, ****kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummax : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
```

(continues on next page)

(continued from previous page)

```
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.max [Similar functionality] but ignores `NaN` values.

DataFrame.max [Return the maximum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cummin (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummin : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.min [Similar functionality] but ignores `NaN` values.

DataFrame.min [Return the minimum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumprod (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, ****kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumprod : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.prod [Similar functionality] but ignores `NaN` values.

DataFrame.prod [Return the product over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumsum(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumsum : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.sum [Similar functionality] but ignores `NaN` values.

DataFrame.sum [Return the sum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

describe (*percentiles=None*, *include=None*, *exclude=None*)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.

- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

summary: Series/DataFrame of summary statistics

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                       3
unique                      2
top       2010-01-01 00:00:00
freq                        2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                     'numeric': [1, 2, 3],
...                     'categorical': pd.Categorical(['d','e','f'])
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
        categorical  numeric object
count             3      3.0      3
unique            3      NaN      3
top               f      NaN      c
freq              1      NaN      1
mean            NaN      2.0    NaN
std             NaN      1.0    NaN
min             NaN      1.0    NaN
25%             NaN      1.5    NaN
50%             NaN      2.0    NaN
75%             NaN      2.5    NaN
max             NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
```

(continues on next page)

(continued from previous page)

```
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
        categorical  numeric
count            3      3.0
unique           3      NaN
top              f      NaN
freq             1      NaN
mean           NaN      2.0
std            NaN      1.0
min            NaN      1.0
25%            NaN      1.5
```

(continues on next page)

(continued from previous page)

```
50%             NaN      2.0
75%             NaN      2.5
max             NaN      3.0
```

DataFrame.count DataFrame.max DataFrame.min DataFrame.mean DataFrame.std DataFrame.select_dtypes

diff (*periods=1*, *axis=0*)

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

periods [int, default 1] Periods to shift for calculating difference, accepts negative values.

axis [{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

New in version 0.16.1..

diffed : DataFrame

Series.diff: First discrete difference for a Series. DataFrame.pct_change: Percent change over given number of periods. DataFrame.shift: Shift index by desired number of periods with an

optional time freq.

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a    b     c
0 NaN  0.0   0.0
1 NaN -1.0   3.0
2 NaN -1.0   7.0
3 NaN -1.0  13.0
4 NaN  0.0  20.0
5 NaN  2.0  28.0
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

div(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

divide(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

dot (*other*)
Matrix multiplication with DataFrame or Series objects. Can also be called using *self @ other* in Python >= 3.5.

other : DataFrame or Series

dot_product : DataFrame or Series

drop (*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)
Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

labels [single label or list-like] Index or column labels to drop.

axis [{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index, columns [single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

New in version 0.21.0.

level [int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace [bool, default False] If True, do operation inplace and return None.

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

dropped : pandas.DataFrame

DataFrame.loc : Label-location based indexer for selection by label. DataFrame.dropna : Return DataFrame with labels on given axis omitted

where (all or any) data are missing

DataFrame.drop_duplicates [Return DataFrame with duplicate rows] removed, optionally only considering certain columns

Series.drop : Return Series with specified index labels removed.

KeyError If none of the labels are found in the selected axis

```
>>> df = pd.DataFrame(np.arange(12).reshape(3,4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                              [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3,0.2]])
>>> df
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
lama    speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

drop_duplicates(*subset=None*, *keep='first'*, *inplace=False*)
Return DataFrame with duplicate rows removed, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Drop duplicates except for the first occurrence.
- `last` : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

inplace [boolean, default False] Whether to drop duplicates in place or to return a copy

deduplicated : DataFrame

dropna(*axis=0*, *how='any'*, *thresh=None*, *subset=None*, *inplace=False*)
Remove missing values.

See the User Guide for more on which values are considered missing, and how to work with missing data.

axis [{0 or 'index', 1 or 'columns'}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Deprecated since version 0.23.0:: Pass tuple or list to drop on multiple

axes.

how [{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

thresh [int, optional] Require that many non-NA values.

subset [array-like, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace [bool, default False] If True, do operation inplace and return None.

DataFrame DataFrame with NA entries dropped from it.

DataFrame.isna: Indicate missing values. DataFrame.notna : Indicate existing (non-missing) values. DataFrame.fillna : Replace missing values. Series.dropna : Drop missing values. Index.dropna : Drop missing indices.

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'born'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type of each column.

pandas.DataFrame.ftypes : dtype and sparsity information.

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
```

(continues on next page)

(continued from previous page)

```
string              object
dtype: object
```

duplicated(*subset=None*, *keep='first'*)

Return boolean Series denoting duplicate rows, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Mark duplicates as `True` except for the first occurrence.
- `last` : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

duplicated : Series

empty

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

bool If DataFrame is empty, return True, if not return False.

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
>>> df.dropna().empty
True
```

pandas.Series.dropna pandas.DataFrame.dropna

eq(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods eq

equals(*other*)

Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

eval(*expr*, *inplace=False*, ***kwargs*)

Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

expr [str] The expression string to evaluate.

inplace [bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

New in version 0.18.0..

kwargs [dict] See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

ndarray, scalar, or pandas object The result of the evaluation.

DataFrame.query [Evaluates a boolean expression to query the columns] of a frame.

DataFrame.assign [Can evaluate an expression or function to create new] values for a column.

pandas.eval [Evaluate a Python expression as a string using various] backends.

For more details see the API documentation for `eval()`. For detailed examples see enhancing performance with eval.

```
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
>>> df.eval('A + B')
0    11
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
```

Use `inplace=True` to modify the original DataFrame.

```
>>> df.eval('C = A + B', inplace=True)
>>> df
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
```

ewm(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*)
Provides exponential weighted functions

New in version 0.18.0.

com [float, optional] Specify decay in terms of center of mass, $\alpha = 1/(1 + com),$ for $com \geq 0$

span [float, optional] Specify decay in terms of span, $\alpha = 2/(span + 1),$ for $span \geq 1$

halflife [float, optional] Specify decay in terms of half-life, $\alpha = 1 - exp(log(0.5)/halflife),$ for $halflife > 0$

alpha [float, optional] Specify smoothing factor α directly, $0 < \alpha \leq 1$

New in version 0.18.0.

min_periods [int, default 0] Minimum number of observations in window required to have a value (otherwise result is NA).

adjust [boolean, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

ignore_na [boolean, default False] Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), . . . , 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

rolling : Provides rolling window calculations expanding : Provides expanding transformations.

expanding(*min_periods=1*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

min_periods [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

center [boolean, default False] Set the labels at the center of the window.

axis : int or string, default 0

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

rolling : Provides rolling window calculations ewm : Provides exponential weighted functions

ffill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)

Synonym for `DataFrame.fillna(method='ffill')`

fillna(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)

Fill NA/NaN values using the specified method

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis : {0 or 'index', 1 or 'columns'} inplace : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

interpolate : Fill NaN values using interpolation. reindex, asfreq

filled : DataFrame

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                    columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 2.0 1
```

(continues on next page)

(continued from previous page)

```
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

filter (*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

items [list-like] List of info axis to restrict to (must not all be present)

like [string] Keep info axis where "arg in col == True"

regex [string (regular expression)] Keep info axis with re.search(regex, col) == True

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame

same type as input object

```
>>> df
one  two  three
mouse     1    2      3
rabbit    4    5      6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
one  two  three
rabbit    4    5      6
```

pandas.DataFrame.loc

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

filter_result (*expressions*)

Filter result based on a list of expressions

Parameters **expressions** (*list((str, comparator, float))*) – A list of triples consisting of (method_name, comparator, threshold)

Returns A new filtered AResult object

Return type *AResult*

first(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

subset : type of caller

last : Select final periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

first_valid_index()

Return index for first non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

floordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rfloordiv

classmethod from_csv(*path*, *header=0*, *sep=','*, *index_col=0*, *parse_dates=True*, *encoding=None*, *tupleize_cols=None*, *infer_datetime_format=False*)

Read CSV file.

Deprecated since version 0.21.0: Use `pandas.read_csv()` instead.

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a DataFrame of time series data.

This method only differs from the preferred `pandas.read_csv()` in some defaults:

- *index_col* is `0` instead of `None` (take first column as index by default)
- *parse_dates* is `True` instead of `False` (try parsing the index as datetime by default)

So a `pd.DataFrame.from_csv(path)` can be replaced by `pd.read_csv(path, index_col=0, parse_dates=True)`.

path : string file path or file handle / StringIO header : int, default 0

Row to use as header (skip prior rows)

sep [string, default ','] Field delimiter

index_col [int or sequence, default 0] Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

parse_dates [boolean, default True] Parse dates. Different default from read_table

tupleize_cols [boolean, default False] write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

infer_datetime_format: boolean, default False If True and *parse_dates* is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

pandas.read_csv

y : DataFrame

classmethod from_dict(*data*, *orient='columns'*, *dtype=None*, *columns=None*)

Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

data [dict] Of the form {field : array-like} or {field : dict}.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'.

dtype [dtype, default None] Data type to force, otherwise infer.

columns [list, default None] Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'`.

New in version 0.23.0.

pandas.DataFrame

DataFrame.from_records [DataFrame from ndarray (structured] dtype), list of tuples, dict, or DataFrame

DataFrame : DataFrame object creation using constructor

By default the keys of the dict become the DataFrame columns:

```
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Specify `orient='index'` to create the DataFrame using dictionary keys as rows:

```
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
       0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                        columns=['A', 'B', 'C', 'D'])
       A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
```

classmethod from_items(*items*, *columns=None*, *orient='columns'*)

Construct a dataframe from a list of tuples

Deprecated since version 0.23.0: *from_items* is deprecated and will be removed in a future version. Use `DataFrame.from_dict(dict(items))` instead. `DataFrame.from_dict(OrderedDict(items))` may be used to preserve the key order.

Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the columns, but depends on the specified orientation). The values should be arrays or Series.

items [sequence of (key, value) pairs] Values should be arrays or Series.

columns [sequence of column labels, optional] Must be passed if orient='index'.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the input correspond to column labels, pass 'columns' (default). Otherwise if the keys correspond to the index, pass 'index'.

frame : DataFrame

classmethod from_records(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float=False*, *nrows=None*)

Convert structured or record ndarray to DataFrame

data : ndarray (structured dtype), list of tuples, dict, or DataFrame index : string, list of fields, array-like

Field of array to use as the index, alternately a specific set of input labels to use

exclude [sequence, default None] Columns or fields to exclude

columns [sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

coerce_float [boolean, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

df : DataFrame

ftypes
Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type and indication of sparse/dense of each column.

pandas.DataFrame.dtypes: Series with just dtype information. pandas.SparseDataFrame : Container for sparse tabular data.

Sparse data should have the same dtypes as its dense representation.

```
>>> import numpy as np
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).ftypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).ftypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

ge(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods ge

get(*key*, *default=None*)
Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

key : object

value : type of items contained in object

get_dtype_counts()
Return counts of unique dtypes in this object.

dtype [Series] Series with the count of columns with each dtype.

dtypes : Return the dtypes in this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
```

(continues on next page)

(continued from previous page)

```
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

get_ftype_counts()
Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

ftypes [Return ftypes (indication of sparse/dense and dtype) in] this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

get_value(*index*, *col*, *takeable=False*)
Quickly retrieve single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label takeable : interpret the index/col as indexers, default False

value : scalar value

get_values()
Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a *pandas.SparseArray*, the data are first converted to a dense representation.

numpy.ndarray Numpy representation of DataFrame

values : Numpy representation of DataFrame. pandas.SparseArray : Container for sparse data.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a      b    c
```

(continues on next page)

(continued from previous page)

```
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
     a    c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

groupby (*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=False*, *observed=False*, ***kwargs*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

by [mapping, function, label, or list of labels] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted a (single) key.

axis : int, default 0 level : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

as_index [boolean, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort [boolean, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.

group_keys [boolean, default True] When calling apply, add group keys to index to identify pieces

squeeze [boolean, default False] reduce the dimensionality of the return type if possible, otherwise return a consistent type

observed [boolean, default False] This only applies if any of the groupers are Categoricals If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

GroupBy object

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

See the user guide for more.

resample [Convenience method for frequency conversion and resampling] of time series.

gt (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods gt

head (*n=5*)
Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

n [int, default 5] Number of rows to select.

obj_head [type of caller] The first *n* rows of the caller object.

pandas.DataFrame.tail: Returns the last *n* rows.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

hist (*column=None*, *by=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *ax=None*, *sharex=False*, *sharey=False*, *figsize=None*, *layout=None*, *bins=10*, ***kwds*)
Make a histogram of the DataFrame's.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

data [DataFrame] The pandas object holding the data.

column [string or sequence] If passed, will be used to limit data to a subset of columns.

by [object, optional] If passed, then used to form histograms for separate groups.

grid [boolean, default True] Whether to show axis grid lines.

xlabelsize [int, default None] If specified changes the x-axis label size.

xrot [float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize [int, default None] If specified changes the y-axis label size.

yrot [float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

ax [Matplotlib axes object, default None] The axes to plot the histogram on.

sharex [boolean, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey [boolean, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize [tuple] The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

layout [tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

axes : matplotlib.AxesSubplot or numpy.ndarray of them

matplotlib.pyplot.hist : Plot a histogram using matplotlib.

iat

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.at : Access a single value for a row/column label pair DataFrame.loc : Access a group of rows and columns by label(s) DataFrame.iloc : Access a group of rows and columns by integer position(s)

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

IndexError When integer position is out of bounds

idxmax (*axis=0*, *skipna=True*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmax : Series

This method is the DataFrame version of `ndarray.argmax`.

Series.idxmax

idxmin (*axis=0*, *skipna=True*)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmin : Series

This method is the DataFrame version of `ndarray.argmin`.

Series.idxmin

iloc

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.

- A boolean array.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at Selection by Position

index
The index (row labels) of the DataFrame.

infer_objects()
Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to numeric typeR

converted : same type as input object

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

info(*verbose=None*, *buf=None*, *max_cols=None*, *memory_usage=None*, *null_counts=None*)
Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

verbose [bool, optional] Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

buf [writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

max_cols [int, optional] When to switch from the verbose to the truncated output. If the DataFrame has more than *max_cols* columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

memory_usage [bool, str, optional] Specifies whether total memory usage of the DataFrame elements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources.

null_counts [bool, optional] Whether to show the non-null counts. By default, this is shown only if the frame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

None This method prints a summary of a DataFrame and returns None.

DataFrame.describe: Generate descriptive statistics of DataFrame columns.

DataFrame.memory_usage: Memory usage of DataFrame columns.

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                    "float_col": float_values})
>>> df
   int_col text_col  float_col
0        1    alpha       0.00
1        2     beta       0.25
2        3    gamma       0.50
3        4    delta       0.75
4        5  epsilon       1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
int_col      5 non-null int64
text_col     5 non-null object
float_col    5 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
```

(continues on next page)

(continued from previous page)

```
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w", encoding="utf-8") as f:
...     f.write(s)
260
```

The *memory_usage* parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 22.9+ MB
```

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 188.8 MB
```

insert (*loc*, *column*, *value*, *allow_duplicates=False*)

Insert column into DataFrame at specified location.

Raises a ValueError if *column* is already contained in the DataFrame, unless *allow_duplicates* is set to True.

loc [int] Insertion index. Must verify 0 <= loc <= len(columns)

column [string, number, or hashable object] label of the inserted column

value : int, Series, or array-like allow_duplicates : bool, optional

interpolate (*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

method [{'linear', 'time', 'index', 'values', 'nearest', 'zero',]

'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default

- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to `scipy.interpolate.interp1d`. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
- 'from_derivatives' refers to BPoly.from_derivatives which replaces 'piecewise_polynomial' interpolation method in scipy 0.18

New in version 0.18.1: Added support for the 'akima' method Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in scipy 0.18; backwards-compatible with scipy < 0.18

axis [{0, 1}, default 0]

- 0: fill column-by-column
- 1: fill row-by-row

limit [int, default None.] Maximum number of consecutive NaNs to fill. Must be greater than 0.

limit_direction : {'forward', 'backward', 'both'}, default 'forward' limit_area : {'inside', 'outside'}, default None

- None: (default) no fill restriction
- 'inside' Only fill NaNs surrounded by valid values (interpolate).
- 'outside' Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

inplace [bool, default False] Update the NDFrame in place if possible.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

kwargs : keyword arguments to pass on to the interpolating function.

Series or DataFrame of same shape interpolated at the NaNs

reindex, replace, fillna

Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

is_copy

isin(*values*)

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

values [iterable, Series, DataFrame or dictionary] The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dictionary, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

DataFrame of booleans

When `values` is a list:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
       A      B
0   True   True
1  False  False
2   True  False
```

When `values` is a dict:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
       A      B
0   True  False  # Note that B didn't match the 1 here.
1  False   True
2   True   True
```

When `values` is a Series or DataFrame:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
       A      B
0   True  False
1  False  False  # Column A in `other` has a 3, but not at index 1.
2   True   True
```

isna()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
```

(continues on next page)

(continued from previous page)

```
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

isnull()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
```

(continues on next page)

(continued from previous page)

```
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

items()
Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iteritems()
Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iterrows()
Iterate over DataFrame rows as (index, Series) pairs.

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
>>> print(df['int'].dtype)
int64
```

   To preserve dtypes while iterating over the rows, it is better to use *itertuples()* which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

it [generator] A generator that iterates over the rows of the frame.

itertuples : Iterate over DataFrame rows as namedtuples of the values. iteritems : Iterate over (column name, Series) pairs.

itertuples(*index=True*, *name='Pandas'*)

Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.

index [boolean, default True] If True, return the index as the first element of the tuple.

name [string, default "Pandas"] The name of the returned namedtuples or None to return regular tuples.

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. iteritems : Iterate over (column name, Series) pairs.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [0.1, 0.2]},
                      index=['a', 'b'])
>>> df
   col1  col2
a     1   0.1
b     2   0.2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='a', col1=1, col2=0.10000000000000001)
Pandas(Index='b', col1=2, col2=0.20000000000000001)
```

ix

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at Advanced Indexing.

join(*other*, *on=None*, *how='left'*, *lsuffix=''*, *rsuffix=''*, *sort=False*)

Join columns with other DataFrame either on index or on a key column. Efficiently Join multiple DataFrame objects by index at once by passing a list.

other [DataFrame, Series with name field set, or list of DataFrame] Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame

on [name, tuple/list of names, or array-like] Column or index level name(s) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

how [{'left', 'right', 'outer', 'inner'}, default: 'left'] How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use other frame's index

- outer: form union of calling frame's index (or column if on is specified) with other frame's index, and sort it lexicographically
- inner: form intersection of calling frame's index (or column if on is specified) with other frame's index, preserving the order of the calling's one

lsuffix [string] Suffix to use from left frame's overlapping columns

rsuffix [string] Suffix to use from right frame's overlapping columns

sort [boolean, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword)

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

Support for specifying index levels as the *on* parameter was added in version 0.23.0

```
>>> caller = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                        'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> caller
    A key
0  A0  K0
1  A1  K1
2  A2  K2
3  A3  K3
4  A4  K4
5  A5  K5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
    B key
0  B0  K0
1  B1  K1
2  B2  K2
```

Join DataFrames using their indexes.

```
>>> caller.join(other, lsuffix='_caller', rsuffix='_other')
```

```
>>>     A key_caller    B key_other
    0  A0          K0   B0         K0
    1  A1          K1   B1         K1
    2  A2          K2   B2         K2
    3  A3          K3  NaN        NaN
    4  A4          K4  NaN        NaN
    5  A5          K5  NaN        NaN
```

If we want to join using the key columns, we need to set key to be the index in both caller and other. The joined DataFrame will have key as its index.

```
>>> caller.set_index('key').join(other.set_index('key'))
```

```
>>>      A    B
    key
    K0   A0   B0
```

(continues on next page)

(continued from previous page)

```
    K1   A1   B1
    K2   A2   B2
    K3   A3  NaN
    K4   A4  NaN
    K5   A5  NaN
```

Another option to join using the key columns is to use the on parameter. DataFrame.join always uses other's index but we can use any column in the caller. This method preserves the original caller's index in the result.

```
>>> caller.join(other.set_index('key'), on='key')
```

```
>>>     A key    B
    0  A0  K0   B0
    1  A1  K1   B1
    2  A2  K2   B2
    3  A3  K3  NaN
    4  A4  K4  NaN
    5  A5  K5  NaN
```

DataFrame.merge : For column(s)-on-columns(s) operations

joined : DataFrame

keys()

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

kurt(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

kurtosis(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

last(*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

subset : type of caller

first : Select initial periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

last_valid_index()

Return index for last non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

le(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods le

loc

Access a group of rows and columns by label(s) or a boolean array.

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

  Warning: Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.

- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at Selection by Label

DataFrame.at : Access a single value for a row/column label pair DataFrame.iloc : Access group of rows and columns by integer position(s) DataFrame.xs : Returns a cross-section (row(s) or column(s)) from the

Series/DataFrame.

Series.loc : Access group of values using labels

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=['cobra', 'viper', 'sidewinder'],
...       columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
```

(continues on next page)

(continued from previous page)

```
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

KeyError: when any items are not found

lookup (*row_labels*, *col_labels*)

Label-based "fancy indexing" function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

row_labels [sequence] The row labels to use for lookup

col_labels [sequence] The column labels to use for lookup

Akin to:

```
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

values [ndarray] The found values

lt (*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods lt

mad(*axis=None*, *skipna=None*, *level=None*)
Return the mean absolute deviation of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mad : Series or DataFrame (if level specified)

mask(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

`DataFrame.where()`

max (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the maximum of the values in the object. If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular

level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

max : Series or DataFrame (if level specified)

mean (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the mean of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mean : Series or DataFrame (if level specified)

median (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the median of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

median : Series or DataFrame (if level specified)

melt (*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *col_level=None*)
"Unpivots" a DataFrame from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

New in version 0.20.0.

frame : DataFrame id_vars : tuple, list, or ndarray, optional

Column(s) to use as identifier variables.

value_vars [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name [scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name [scalar, default 'value'] Name to use for the 'value' column.

col_level [int or string, optional] If columns are a MultiIndex then use this level to melt.

melt pivot_table DataFrame.pivot

```
>>> import pandas as pd
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> df.melt(id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

memory_usage(*index=True*, *deep=False*)

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

index [bool, default True] Specifies whether to include the memory usage of the DataFrame's index in returned Series. If `index=True` the memory usage of the index the first item in the output.

deep [bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

sizes [Series] A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

numpy.ndarray.nbytes [Total bytes consumed by the elements of an] ndarray.

Series.memory_usage : Bytes consumed by a Series. pandas.Categorical : Memory-efficient array for string values with

many repeated values.

DataFrame.info : Concise summary of a DataFrame.

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64  complex128 object  bool
0      1      1.0      (1+0j)      1  True
1      1      1.0      (1+0j)      1  True
2      1      1.0      (1+0j)      1  True
3      1      1.0      (1+0j)      1  True
4      1      1.0      (1+0j)      1  True
```

```
>>> df.memory_usage()
Index            80
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index             80
int64          40000
float64        40000
complex128     80000
object        160000
bool            5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5168
```

merge(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *sort=False*, *suffixes=('_x', '_y')*, *copy=True*, *indicator=False*, *validate=None*)

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

right : DataFrame how : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys

on [label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [boolean, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

right_index [boolean, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index

sort [boolean, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword)

suffixes [2-length sequence (tuple, list, ...)] Suffix to apply to overlapping column names in the left and right side, respectively

copy [boolean, default True] If False, do not copy data unnecessarily

indicator [boolean or string, default False] If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row

will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in 'left' DataFrame, "right_only" for observations whose merge key only appears in 'right' DataFrame, and "both" if the observation's merge key is found in both.

validate [string, default None] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

New in version 0.21.0.

Support for specifying index levels as the *on*, *left_on*, and *right_on* parameters was added in version 0.23.0

```
>>> A              >>> B
    lkey value         rkey value
0   foo  1         0   foo  5
1   bar  2         1   bar  6
2   baz  3         2   qux  7
3   foo  4         3   bar  8
```

```
>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0  foo   1        foo   5
1  foo   4        foo   5
2  bar   2        bar   6
3  bar   2        bar   8
4  baz   3        NaN   NaN
5  NaN   NaN      qux   7
```

merged [DataFrame] The output type will the be same as 'left', if it is a subclass of DataFrame.

merge_ordered merge_asof DataFrame.join

merge_results(*others*)

Merges results of the same type and returns a merged result

Parameters **others** (list(*AResult*)/*AResult*) – A (list of) *AResult* object(s) of the same class

Returns A new merged *AResult* object

Return type *AResult*

min(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the minimum of the values in the object. If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min : Series or DataFrame (if level specified)

mod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmod

mode(*axis=0*, *numeric_only=False*)
Gets the mode(s) of each element along the axis selected. Adds a row for each mode per label, fills in gaps with nan.

Note that there could be multiple values returned for the selected axis (when more than one item share the maximum frequency), which is the reason why a dataframe is returned. If you want to impute missing values with the mode in a dataframe `df`, you can just do this: `df.fillna(df.mode().iloc[0])`

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index' : get mode of each column
- 1 or 'columns' : get mode of each row

numeric_only [boolean, default False] if True, only apply to numeric columns

modes : DataFrame (sorted)

```
>>> df = pd.DataFrame({'A': [1, 2, 1, 2, 1, 2, 3]})
>>> df.mode()
   A
0  1
1  2
```

mul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

multiply (*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

ndim
Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

ndarray.ndim

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

ne (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods ne

nlargest (*n*, *columns*, *keep='first'*)
Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant.

n [int] Number of rows to return.

columns [label or list of labels] Column label(s) to order by.

keep [{'first', 'last'}, default 'first'] Where there are duplicate values:

- *first* : prioritize the first occurrence(s)
- *last* : prioritize the last occurrence(s)

DataFrame The first *n* rows ordered by the given columns in descending order.

DataFrame.nsmallest [Return the first *n* rows ordered by *columns* in] ascending order.

DataFrame.sort_values : Sort DataFrame by the values DataFrame.head : Return the first *n* rows without re-ordering.

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 10, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df
    a  b    c
0   1  a  1.0
1  10  b  2.0
2   8  d  NaN
3  10  c  3.0
4  -1  e  4.0
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column "a".

```
>>> df.nlargest(3, 'a')
    a  b    c
1  10  b  2.0
3  10  c  3.0
2   8  d  NaN
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'a', keep='last')
    a  b    c
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

To order by the largest values in column "a" and then "c", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['a', 'c'])
    a  b    c
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

Attempting to use `nlargest` on non-numeric dtypes will raise a `TypeError`:

```
>>> df.nlargest(3, 'b')
Traceback (most recent call last):
TypeError: Column 'b' has dtype object, cannot use method 'nlargest'
```

notna()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

notnull()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set

pandas.options.mode.use_inf_as_na = True). NA values, such as None or numpy.NaN, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

nsmallest (*n*, *columns*, *keep='first'*)

Get the rows of a DataFrame sorted by the *n* smallest values of *columns*.

n [int] Number of items to retrieve

columns [list or str] Column name or names to order by

keep [{'first', 'last'}, default 'first'] Where there are duplicate values: - first : take the first occurrence. - last : take the last occurrence.

DataFrame

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 11, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nsmallest(3, 'a')
```

(continues on next page)

(continued from previous page)

```
   a  b   c
4 -1  e   4
0  1  a   1
2  8  d NaN
```

nunique(*axis=0*, *dropna=True*)

Return Series with number of distinct observations over requested axis.

New in version 0.20.0.

axis : {0 or 'index', 1 or 'columns'}, default 0 dropna : boolean, default True

Don't include NaN in the counts.

nunique : Series

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 1, 1]})
>>> df.nunique()
A    3
B    1
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
```

pct_change(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs** Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

chg [Series or DataFrame] The same type as the calling object.

Series.diff : Compute the difference of two elements in a Series. DataFrame.diff : Compute the difference of two elements in a DataFrame. Series.shift : Shift the index by some number of periods. DataFrame.shift : Shift the index by some number of periods.

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...      'FR': [4.0405, 4.0963, 4.3149],
...      'GR': [1.7246, 1.7482, 1.8519],
...      'IT': [804.74, 810.01, 860.13]},
...      index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                   FR        GR        IT
1980-01-01        NaN       NaN       NaN
1980-02-01   0.013810  0.013684  0.006549
1980-03-01   0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...      '2016': [1769950, 30586265],
...      '2015': [1500923, 40912316],
```

(continues on next page)

(continued from previous page)

```
...      '2014': [1371819, 41403351]},
...      index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pipe (*func*, **args*, ***kwargs*)

Apply func(self, *args, **kwargs)

func [function] function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the NDFrame.

args [iterable, optional] positional arguments passed into `func`.

kwargs [mapping, optional] a dictionary of keyword arguments passed into `func`.

object : the return type of `func`.

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pandas.DataFrame.apply pandas.DataFrame.applymap pandas.Series.map

pivot (*index=None*, *columns=None*, *values=None*)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the User Guide for more on reshaping.

index [string or object, optional] Column to use to make new frame's index. If None, uses existing index.

columns [string or object] Column to use to make new frame's columns.

values [string, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Changed in version 0.23.0: Also accept list of column names.

DataFrame Returns reshaped DataFrame.

ValueError: When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

DataFrame.pivot_table [generalization of pivot that can handle] duplicate values for one index/column pair.

DataFrame.unstack [pivot based on the index values instead of a] column.

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
```

(continues on next page)

(continued from previous page)

```
...                    "baz": [1, 2, 3, 4]})
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

pivot_table(*values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*, *dropna=True*, *margins_name='All'*)

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame

values : column to aggregate, optional index : column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

columns [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

aggfunc [function, list of functions, dict, default numpy.mean] If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions

fill_value [scalar, default None] Value to replace missing values with

margins [boolean, default False] Add all row / columns (e.g. for subtotal / grand totals)

dropna [boolean, default True] Do not include columns whose entries are all NaN

margins_name [string, default 'All'] Name of the row / column that will contain the totals when margins is True.

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
>>> df
     A    B      C  D
0  foo  one  small  1
1  foo  one  large  2
2  foo  one  large  2
```

(continues on next page)

(continued from previous page)

```
3  foo  two  small  3
4  foo  two  small  3
5  bar  one  large  4
6  bar  one  small  5
7  bar  two  small  6
8  bar  two  large  7
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': [min, max, np.mean]})
>>> table
                  D   E
               mean max median min
A   C
bar large  5.500000  16   14.5  13
    small  5.500000  15   14.5  14
foo large  2.000000  10    9.5   9
    small  2.333333  12   11.0   8
```

table : DataFrame

DataFrame.pivot [pivot without aggregation that can handle] non-numeric data

plot

alias of `pandas.plotting._core.FramePlotMethods`

pop(*item*)

Return item and drop from frame. Raise KeyError if not found.

item [str] Column label to be popped

popped : Series

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
```

(continues on next page)

(continued from previous page)

```
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

pow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rpow

prod(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

product (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

quantile(*q=0.5*, *axis=0*, *numeric_only=True*, *interpolation='linear'*)

Return values at the given quantile over requested axis, a la numpy.percentile.

q [float or array-like, default 0.5 (50% quantile)] 0 <= q <= 1, the quantile(s) to compute

axis [{0, 1, 'index', 'columns'} (default 0)] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

numeric_only [boolean, default True] If False, the quantile of datetime and timedelta data will be computed as well

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.18.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

quantiles : Series or DataFrame

- If `q` is an array, a DataFrame will be returned where the index is `q`, the columns are the columns of self, and the values are the quantiles.
- If `q` is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                      columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
                       'B': [pd.Timestamp('2010'),
                             pd.Timestamp('2011')],
                       'C': [pd.Timedelta('1 days'),
                             pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
```

(continues on next page)

(continued from previous page)

```
C        1 days 12:00:00
Name: 0.5, dtype: object
```

pandas.core.window.Rolling.quantile

query(*expr*, *inplace=False*, ***kwargs*)

Query the columns of a frame with a boolean expression.

expr [string] The query string to evaluate. You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

inplace [bool] Whether the query should modify the data in place or return a modified copy

New in version 0.18.0.

kwargs [dict] See the documentation for `pandas.eval()` for complete details on the keyword arguments accepted by `DataFrame.query()`.

q : DataFrame

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level `pandas.eval()` function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the `query` documentation in indexing.

pandas.eval DataFrame.eval

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = pd.DataFrame(randn(10, 2), columns=list('ab'))
>>> df.query('a > b')
>>> df[df.a > df.b]  # same result as the previous expression
```

radd(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.add

rank (*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)
Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values

axis [{0 or 'index', 1 or 'columns'}, default 0] index to direct ranking

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

ranks : same type as caller

rdiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

reindex(***kwargs*)
Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like, optional] New labels / index to conform the axis specified by 'axis' to.

index, columns [array-like, optional (should be specified using keywords)] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis [int or str, optional] Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...       'http_status': [200,200,404,404,301],
...       'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...        index=index)
>>> df
            http_status  response_time
Firefox             200           0.04
Chrome              200           0.02
Safari              404           0.07
IE10                404           0.08
Konqueror           301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
```

(continues on next page)

(continued from previous page)

```
IE10                  404          0.08
Chrome                200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to backpropagate the last valid value to fill the NaN values, pass bfill as an argument to the method keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29     100
2009-12-30     100
2009-12-31     100
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

Please note that the NaN value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the NaN values present in the original dataframe, use the fillna() method.

See the user guide for more.

reindexed : DataFrame

reindex_axis (*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=nan*)

Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis : {0 or 'index', 1 or 'columns'} method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional

Method to use for filling holes in reindexed DataFrame:

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation abs(index[indexer] - target) <= tolerance.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

reindex, reindex_like

reindexed : DataFrame

reindex_like(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)
Return an object with matching indices to myself.

other : Object method : string or None copy : boolean, default True limit : int, default None

Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

reindexed : same as input

rename(***kwargs*)
Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the user guide for more.

mapper, index, columns [dict-like or function, optional] dict-like or functions transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

axis [int or str, optional] Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

copy [boolean, default True] Also copy underlying data

inplace [boolean, default False] Whether to return a new DataFrame. If True then value of copy is ignored.

level [int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

renamed : DataFrame

pandas.DataFrame.rename_axis

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
```

(continues on next page)

(continued from previous page)

```
1  2  5
2  3  6
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

rename_axis (*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter the name of the index or columns.

mapper [scalar, list-like, optional] Value to set as the axis name attribute.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis.

copy [boolean, default True] Also copy underlying data.

inplace [boolean, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

renamed [Series, DataFrame, or None] The same type as the caller or None if *inplace* is True.

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

pandas.Series.rename : Alter Series index labels or name pandas.DataFrame.rename : Alter DataFrame index labels or name pandas.Index.rename : Set new names on index

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s.rename_axis("foo")
foo
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")
     A  B
foo
0    1  4
1    2  5
2    3  6
```

```
>>> df.rename_axis("bar", axis="columns")
bar  A  B
0    1  4
1    2  5
2    3  6
```

reorder_levels (*order*, *axis=0*)

Rearrange index levels using input order. May not drop or duplicate levels

order [list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).

axis [int] Where to reorder levels.

type of caller (new object)

replace (*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)

Replace values given in *to_replace* with *value*.

Values of the DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

to_replace [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be `None` to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:

- This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

value [scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace [boolean, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

regex [bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method [{'pad', 'ffill', 'bfill', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

DataFrame.fillna : Fill NA values DataFrame.where : Replace values based on boolean condition Series.str.replace : Simple string replacement.

DataFrame Object after replacement.

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Scalar ‘to_replace‘ and ‘value‘

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like ‘to_replace‘

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
2    3
3    3
4    4
dtype: int64
```

dict-like ‘to_replace‘

```
>>> df.replace({0: 10, 1: 100})
     A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
     A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
     A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression 'to_replace'

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
      A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$':'new', 'foo':'xyz'})
      A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
      A    B
0   new  abc
1   new  new
2  bait  xyz
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
```

(continues on next page)

(continued from previous page)

```
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the pecularities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value=None` and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

resample(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

rule [string] the offset string or object representing target conversion

axis : int, optional, default 0 closed : {'right', 'left'}

Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label [{'right', 'left'}] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention [{'start', 'end', 's', 'e'}] For PeriodIndex only, controls whether to use the start or end of *rule*

kind: {'timestamp', 'period'}, optional Pass 'timestamp' to convert the resulting index to a `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`. By default the input representation is retained.

loffset [timedelta] Adjust the resampled time labels

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

on [string, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [string or int, optional] For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

Resampler object

See the user guide for more.

To learn more about the offset strings, please see this link.

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5] #select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like)+5
```

```
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))
>>> s
2012    1
```

(continues on next page)

(continued from previous page)

```
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using 'start' *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
                     a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                       columns=['a', 'b', 'c', 'd'],
                       index=pd.MultiIndex.from_product([time, [1, 2]])
                       )
>>> df2.resample('3T', level=0).sum()
                     a  b   c   d
2000-01-01 00:00:00  0  6  12  18
2000-01-01 00:03:00  0  4   8  12
```

groupby : Group by mapping, function, label, or list of labels.

reset_index(*level=None*, *drop=False*, *inplace=False*, *col_level=0*, *col_fill=''*)

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level_0' (if 'index' is already taken) will be used.

level [int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default

drop [boolean, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

col_level [int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill [object, default ''] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

resetted : DataFrame

```
>>> df = pd.DataFrame([('bird',    389.0),
...                    ('bird',     24.0),
...                    ('mammal',   80.5),
...                    ('mammal', np.nan)],
...                   index=['falcon', 'parrot', 'lion', 'monkey'],
...                   columns=('class', 'max_speed'))
>>> df
         class  max_speed
falcon    bird      389.0
parrot    bird       24.0
lion    mammal       80.5
monkey  mammal        NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
    class  max_speed
0    bird      389.0
1    bird       24.0
2  mammal       80.5
3  mammal        NaN
```

You can also use *reset_index* with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                    ('bird', 'parrot'),
...                                    ('mammal', 'lion'),
...                                    ('mammal', 'monkey')],
```

(continues on next page)

(continued from previous page)

```
...                                       names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                      ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
...                    ( 24.0, 'fly'),
...                    ( 80.5, 'run'),
...                    (np.nan, 'jump')],
...                   index=index,
...                   columns=columns)
>>> df
               speed species
                 max    type
class  name
bird   falcon  389.0     fly
       parrot   24.0     fly
mammal lion     80.5     run
       monkey    NaN    jump
```

If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
         class  speed species
                  max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
                speed species
         class    max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

When the index is inserted under another level, we can specify under which one with the parameter *col_fill*:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
              species  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

If we specify a nonexistent level for *col_fill*, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
                genus  speed species
                class    max    type
```

(continues on next page)

(continued from previous page)

```
name
falcon          bird  389.0     fly
parrot          bird   24.0     fly
lion          mammal   80.5     run
monkey        mammal    NaN    jump
```

rfloordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.floordiv

rmod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mod

rmul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mul

rolling(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculations.

New in version 0.18.0.

window [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. This is new in 0.19.0

min_periods [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

center [boolean, default False] Set the labels at the center of the window.

win_type [string, default None] Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

on [string, optional] For a DataFrame, column on which to calculate the rolling window, rather than the index

closed [string, default None] Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

axis : int or string, default 0

a Window or Rolling sub-classed for the particular operation

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
     B
0  NaN
1  1.0
```

(continues on next page)

(continued from previous page)

```
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                   index = [pd.Timestamp('20130101 09:00:00'),
...                            pd.Timestamp('20130101 09:00:02'),
...                            pd.Timestamp('20130101 09:00:03'),
...                            pd.Timestamp('20130101 09:00:05'),
...                            pd.Timestamp('20130101 09:00:06')])
```

```
>>> df
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```
>>> df.rolling('2s').sum()
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

To learn more about the offsets & frequency strings, please see this link.

The recognized win_types are:

- boxcar
- triang
- blackman
- hamming
- bartlett
- parzen
- bohman
- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general_gaussian (needs power, width)
- slepian (needs width).

If win_type=None all points are evenly weighted. To learn more about different window types see scipy.signal window functions.

expanding : Provides expanding transformations. ewm : Provides exponential weighted functions

round(*decimals=0*, **args*, ***kwargs*)

Round a DataFrame to a variable number of decimal places.

decimals [int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

```
>>> df = pd.DataFrame(np.random.random([3, 3]),
...     columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
>>> df
                A         B         C
first    0.028208  0.992815  0.173891
second   0.038683  0.645646  0.577595
third    0.877076  0.149370  0.491027
>>> df.round(2)
           A     B     C
first   0.03  0.99  0.17
second  0.04  0.65  0.58
third   0.88  0.15  0.49
>>> df.round({'A': 1, 'C': 2})
          A         B     C
first   0.0  0.992815  0.17
second  0.0  0.645646  0.58
third   0.9  0.149370  0.49
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
          A  B     C
first   0.0  1  0.17
```

(continues on next page)

(continued from previous page)

```
second  0.0  1  0.58
third   0.9  0  0.49
```

DataFrame object

numpy.around Series.round

rpow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.pow

rsub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
```

(continues on next page)

(continued from previous page)

```
...                    index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.sub

rtruediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*)

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

n [int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac [float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace [boolean, optional] Sample with or without replacement. Default = False.

weights [str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. inf and -inf values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

A new object of same type as caller.

Generate an example `Series` and `DataFrame`:

```
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0   -0.038497
1    1.820773
2   -0.972766
3   -1.598270
4   -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
          A         B         C         D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661
```

Next extract a random sample from both of these objects...

3 random elements from the `Series`:

```
>>> s.sample(n=3)
27   -0.994689
55   -1.049016
67   -0.224565
dtype: float64
```

And a random 10% of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.1, replace=True)
           A         B         C         D
35  1.981780  0.142106  1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640  0.217116
40  0.823173 -0.078816  1.009536  1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
6  -0.148339  0.832938  1.787600 -1.383767
```

You can use *random state* for reproducibility:

```
>>> df.sample(random_state=1)
A         B         C         D
37 -2.027662  0.103611  0.237496 -0.165867
43 -0.259323 -0.583426  1.516140 -0.479118
12 -1.686325 -0.579510  0.985195 -0.460286
8   1.167946  0.429082  1.215742 -1.636041
9   1.197475 -0.864188  1.554031 -1.505264
```

select (*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

crit [function] To be called on each index (label). Should return True or False

axis : int

selection : type of caller

select_dtypes(*include=None*, *exclude=None*)

Return a subset of the DataFrame's columns based on the column dtypes.

include, exclude [scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

ValueError

- If both of `include` and `exclude` are empty
- If `include` and `exclude` have overlapping elements
- If any kind of string dtype is passed in.

subset [DataFrame] The subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.

- To select all *numeric* types, use `np.number` or `'number'`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the numpy dtype hierarchy
- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`
- To select timedeltas, use `np.timedelta64`, `'timedelta'` or `'timedelta64'`
- To select Pandas categorical dtypes, use `'category'`
- To select Pandas datetimetz dtypes, use `'datetimetz'` (new in 0.20.0) or `'datetime64[ns, tz]'`

```
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
        a      b  c
0       1   True  1.0
1       2  False  2.0
2       1   True  1.0
3       2  False  2.0
4       1   True  1.0
5       2  False  2.0
```

```
>>> df.select_dtypes(include='bool')
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

```
>>> df.select_dtypes(include=['float64'])
   c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int'])
       b    c
0   True  1.0
1  False  2.0
2   True  1.0
3  False  2.0
4   True  1.0
5  False  2.0
```

sem(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

sem : Series or DataFrame (if level specified)

set_axis(*labels*, *axis=0*, *inplace=None*)
Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

labels [list-like, Index] The values for the new index.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

pandas.DataFrame.rename_axis : Alter the name of the index or columns.

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

set_index (*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

keys : column label or list of column labels / arrays drop : boolean, default True

Delete columns to be used as the new index

append [boolean, default False] Whether to append columns to existing index

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

verify_integrity [boolean, default False] Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale':[55, 40, 84, 31]})
   month  sale  year
0  1      55    2012
1  4      40    2014
2  7      84    2013
3  10     31    2014
```

Set the index to become the 'month' column:

```
>>> df.set_index('month')
       sale  year
month
1      55    2012
4      40    2014
7      84    2013
10     31    2014
```

Create a multi-index using columns 'year' and 'month':

```
>>> df.set_index(['year', 'month'])
            sale
year  month
2012  1     55
2014  4     40
2013  7     84
2014  10    31
```

Create a multi-index using a set of values and a column:

```
>>> df.set_index([[1, 2, 3, 4], 'year'])
         month  sale
   year
1  2012  1      55
2  2014  4      40
3  2013  7      84
4  2014  10     31
```

dataframe : DataFrame

set_value(*index*, *col*, *value*, *takeable=False*)

Put single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label value : scalar value takeable : interpret the index/col as indexers, default False

frame [DataFrame] If label pair is contained, will be reference to calling DataFrame, otherwise a new object

shape
Return a tuple representing the dimensionality of the DataFrame.

ndarray.shape

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

shift (*periods=1*, *freq=None*, *axis=0*)
Shift index by desired number of periods with an optional time freq

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, optional] Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

axis : {0 or 'index', 1 or 'columns'}

If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data.

shifted : DataFrame

size
Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

ndarray.size

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

skew (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

skew : Series or DataFrame (if level specified)

slice_shift (*periods=1*, *axis=0*)
Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

periods [int] Number of periods to move, can be positive or negative

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

shifted : same type as caller

sort_index (*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)
Sort object by labels (along an axis)

axis : index, columns to direct sorting level : int or level name or list of ints or list of level names

if not None, sort on values in specified index level(s)

ascending [boolean, default True] Sort ascending vs. descending

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining [bool, default True] if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

sorted_obj : DataFrame

sort_values (*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)
Sort by the values along either axis

by [str or list of str] Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels

Changed in version 0.23.0: Allow specifying index or column level names.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis to be sorted

ascending [bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end

sorted_obj : DataFrame

```
>>> df = pd.DataFrame({
...     'col1' : ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2' : [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
```

(continues on next page)

(continued from previous page)

```
... })
>>> df
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
3   NaN  8    4
4   D    7    2
5   C    4    3
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
    col1 col2 col3
1   A    1    1
0   A    2    0
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
    col1 col2 col3
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
3   NaN  8    4
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
    col1 col2 col3
3   NaN  8    4
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
```

sortlevel (*level=0*, *axis=0*, *ascending=True*, *inplace=False*, *sort_remaining=True*)

Sort multilevel index by chosen axis and primary level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order).

Deprecated since version 0.20.0: Use `DataFrame.sort_index()`

level : int axis : {0 or 'index', 1 or 'columns'}, default 0 ascending : boolean, default True inplace : boolean, default False

Sort the DataFrame without creating a new instance

sort_remaining [boolean, default True] Sort by the other levels too.

sorted : DataFrame

DataFrame.sort_index(level=...)

squeeze (*axis=None*)

Squeeze length 1 dimensions.

axis [None, integer or string axis name, optional] The axis to squeeze if 1-sized.

New in version 0.20.0.

scalar if 1-sized, else original object

stack (*level=-1*, *dropna=True*)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

The new index levels are sorted.

level [int, str, list, default -1] Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

dropna [bool, default True] Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of index and column values that are missing from the original dataframe. See Examples section.

DataFrame or Series Stacked dataframe or series.

DataFrame.unstack [Unstack prescribed level(s) from index axis] onto column axis.

DataFrame.pivot [Reshape dataframe from long format to wide] format.

DataFrame.pivot_table [Create a spreadsheet-style pivot table] as a DataFrame.

The function is named by analogy with a collection of books being re-organised from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of of each other (in the index of the dataframe).

Single level columns

```
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
     weight height
cat       0      1
dog       2      3
>>> df_single_level_cols.stack()
cat  weight    0
     height    1
dog  weight    2
     height    3
dtype: int64
```

Multi level columns: simple case

```
>>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol1)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
     weight
         kg    pounds
cat       1         2
dog       2         4
>>> df_multi_level_cols1.stack()
            weight
cat kg           1
    pounds       2
dog kg           2
    pounds       4
```

Missing values

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
    weight height
        kg      m
cat    1.0    2.0
dog    3.0    4.0
>>> df_multi_level_cols2.stack()
        height  weight
cat kg     NaN     1.0
    m      2.0     NaN
dog kg     NaN     3.0
    m      4.0     NaN
```

Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0)
             kg    m
cat height  NaN  2.0
    weight  1.0  NaN
dog height  NaN  4.0
    weight  3.0  NaN
>>> df_multi_level_cols2.stack([0, 1])
cat  height  m     2.0
     weight  kg    1.0
dog  height  m     4.0
     weight  kg    3.0
dtype: float64
```

Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
    weight height
        kg      m
cat    NaN    1.0
dog    2.0    3.0
>>> df_multi_level_cols3.stack(dropna=False)
        height  weight
cat kg     NaN     NaN
    m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
>>> df_multi_level_cols3.stack(dropna=True)
        height  weight
cat m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
```

std(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

std : Series or DataFrame (if level specified)

style
Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

pandas.io.formats.style.Styler

sub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

subtract(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the sum of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

sum : Series or DataFrame (if level specified)

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

swapaxes (*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately

y : same as input

swaplevel (*i=-2*, *j=-1*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

swapped : type of caller (new object)

Changed in version 0.18.1: The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

tail (*n=5*)

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

n [int, default 5] Number of rows to select.

type of caller The last *n* rows of the caller object.

pandas.DataFrame.head : The first *n* rows of the caller object.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

take (*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

indices [array-like] An array of ints indicating which positions to take.

axis [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

taken [type of caller] An array-like containing the elements taken from the object.

DataFrame.loc : Select a subset of a DataFrame by labels. DataFrame.iloc : Select a subset of a DataFrame by positions. numpy.take : Take elements from an array along an axis.

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                    columns=['name', 'class', 'max_speed'],
...                    index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

to_clipboard(*excel=True*, *sep=None*, ***kwargs*)

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

excel [bool, default True]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

sep [str, default `'\t'`] Field delimiter.

****kwargs** These parameters will be passed to DataFrame.to_csv.

DataFrame.to_csv [Write a DataFrame to a comma-separated values] (csv) file.

read_clipboard : Read text from clipboard and pass to read_table.

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

to_csv(*path_or_buf=None*, *sep=', '*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *encoding=None*, *compression=None*, *quoting=None*, *quotechar='"'*, *line_terminator='\n'*, *chunksize=None*, *tupleize_cols=None*, *date_format=None*, *doublequote=True*, *escapechar=None*, *decimal='.'*)

Write DataFrame to a comma-separated values (csv) file

path_or_buf [string or file handle, default None] File path or object, if None is provided the result is returned as a string.

sep [character, default ','] Field delimiter for the output file.

na_rep [string, default ''] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R

mode [str] Python write mode, default 'w'

encoding [string, optional] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

compression [string, optional] A string representing the compression to use in the output file. Allowed values are 'gzip', 'bz2', 'zip', 'xz'. This input is only used when the first argument is a filename.

line_terminator [string, default `'\n'`] The newline character or character sequence to use in the output file

quoting [optional constant from csv module] defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric

quotechar [string (length 1), default '"'] character used to quote fields

doublequote [boolean, default True] Control quoting of *quotechar* inside a field

escapechar [string (length 1), default None] character used to escape *sep* and *quotechar* when appropriate

chunksize [int or None] rows to write at a time

tupleize_cols [boolean, default False] Deprecated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

Write MultiIndex columns as a list of tuples (if True) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if False).

date_format [string, default None] Format string for datetime objects

decimal: string, default '.' Character recognized as decimal separator. E.g. use ',' for European data

to_dense()
Return dense representation of NDFrame (as opposed to sparse)

to_dict(*orient='dict'*, *into=<type 'dict'>*)
Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

orient [str {'dict', 'list', 'series', 'split', 'records', 'index'}] Determines the type of the values of the dictionary.

- 'dict' (default) : dict like {column -> {index -> value}}
- 'list' : dict like {column -> [values]}
- 'series' : dict like {column -> Series(values)}
- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'records' : list like [{column -> value}, ... , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

into [class, default dict] The collections.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

New in version 0.21.0.

result : collections.Mapping like {column -> {index -> value}}

DataFrame.from_dict: create a DataFrame from a dictionary DataFrame.to_json: convert a DataFrame to JSON format

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   col1  col2
a     1   0.50
b     2   0.75
>>> df.to_dict()
{'col1': {'a': 1, 'b': 2}, 'col2': {'a': 0.5, 'b': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': a    1
         b    2
         Name: col1, dtype: int64,
 'col2': a    0.50
         b    0.75
         Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['a', 'b'], 'columns': ['col1', 'col2'],
 'data': [[1.0, 0.5], [2.0, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1.0, 'col2': 0.5}, {'col1': 2.0, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'a': {'col1': 1.0, 'col2': 0.5}, 'b': {'col1': 2.0, 'col2': 0.75}}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('a', 1), ('b', 2)])),
             ('col2', OrderedDict([('a', 0.5), ('b', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1.0, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2.0, 'col2': 0.75})]
```

to_excel(*excel_writer*, *sheet_name='Sheet1'*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *startrow=0*, *startcol=0*, *engine=None*, *merge_cells=True*, *encoding=None*, *inf_rep='inf'*, *verbose=True*, *freeze_panes=None*)

Write DataFrame to an excel sheet

excel_writer [string or ExcelWriter object] File path or existing ExcelWriter

sheet_name [string, default 'Sheet1'] Name of sheet which will contain DataFrame

na_rep [string, default ''] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow : upper left cell row to dump data frame

startcol : upper left cell column to dump data frame

engine [string, default None] write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

merge_cells [boolean, default True] Write MultiIndex and Hierarchical Rows as merged cells.

encoding: string, default None encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep [string, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel)

freeze_panes [tuple of integer (length 2), default None] Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

For compatibility with to_csv, to_excel serializes lists and dicts to strings before writing.

to_feather(*fname*)
write out the binary feather-format for DataFrames

New in version 0.20.0.

fname [str] string file path

to_gbq(*destination_table*, *project_id*, *chunksize=None*, *verbose=None*, *reauth=False*, *if_exists='fail'*, *private_key=None*, *auth_local_webserver=False*, *table_schema=None*)
Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

Authentication to the Google BigQuery service is via OAuth 2.0.

- If `private_key` is provided, the library loads the JSON service account credentials and uses those to authenticate.
- If no `private_key` is provided, the library tries application default credentials.
- If application default credentials are not found or cannot be used with BigQuery, the library authenticates with user account credentials. In this case, you will be asked to grant permissions for product name 'pandas GBQ'.

destination_table [str] Name of table to be written, in the form 'dataset.tablename'.

project_id [str] Google BigQuery Account project ID.

chunksize [int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

reauth [bool, default False] Force Google BigQuery to reauthenticate the user. This is useful if multiple accounts are used.

if_exists [str, default 'fail'] Behavior when the destination table exists. Value can be one of:

'fail' If table exists, do nothing.

'replace' If table exists, drop it, recreate it, and insert data.

'append' If table exists, insert data. Create if does not exist.

private_key [str, optional] Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

auth_local_webserver [bool, default False] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

table_schema [list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'},...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

New in version 0.3.1 of pandas-gbq.

verbose [boolean, deprecated] *Deprecated in Pandas-GBQ 0.4.0.* Use the logging module to adjust verbosity instead.

pandas_gbq.to_gbq : This function in the pandas-gbq library. pandas.read_gbq : Read a DataFrame from Google BigQuery.

to_hdf (*path_or_buf*, *key*, ***kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the user guide.

path_or_buf [str or pandas.HDFStore] File path or HDFStore object.

key [str] Identifier for the group in the store.

mode [{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

format [{'fixed', 'table'}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

append [bool, default False] For Table formats, append the input data to the existing.

data_columns [list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See io.hdf5-query-data-columns. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

errors [str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

DataFrame.read_hdf : Read from HDF file. DataFrame.to_parquet : Write a DataFrame to the binary parquet format. DataFrame.to_sql : Write to a sql table. DataFrame.to_feather : Write out feather-format for DataFrames. DataFrame.to_csv : Write out to a csv file.

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

to_html (*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *bold_rows=True*, *classes=None*, *escape=True*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*, *notebook=False*, *decimal='.'*, *border=None*, *table_id=None*)

Render a DataFrame as an HTML table.

to_html-specific options:

bold_rows [boolean, default True] Make the row labels bold in the output

classes [str or list or tuple, default None] CSS class(es) to apply to the resulting html table

escape [boolean, default True] Convert the characters <, >, and & to HTML-safe sequences.

max_rows [int, optional] Maximum number of rows to show before truncating. If None, show all.

max_cols [int, optional] Maximum number of columns to show before truncating. If None, show all.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe

New in version 0.18.0.

border [int] A `border=border` attribute is included in the opening <*table*> tag. Default `pd.options.html.border`.

New in version 0.19.0.

table_id [str, optional] A css id is included in the opening <*table*> tag if specified.

New in version 0.23.0.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] whether to print column labels, default True

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_json(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression=None*, *index=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

path_or_buf [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'

  - allowed values are: {‘split’,’records’,’index’}
- DataFrame
  - default is ‘columns’
  - allowed values are: {‘split’,’records’,’index’,’columns’,’values’}
- The format of the JSON string
  - ‘split’ : dict like {‘index’ -> [index], ‘columns’ -> [columns], ‘data’ -> [values]}
  - ‘records’ : list like [{column -> value}, … , {column -> value}]
  - ‘index’ : dict like {index -> {column -> value}}
  - ‘columns’ : dict like {column -> {index -> value}}
  - ‘values’ : just the values array
  - ‘table’ : dict like {‘schema’: {schema}, ‘data’: {data}} describing the data, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

date_format [{None, ‘epoch’, ‘iso’}] Type of date conversion. ‘epoch’ = epoch milliseconds, ‘iso’ = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is ‘iso’. For all other orients, the default is ‘epoch’.

double_precision [int, default 10] The number of decimal places to use when encoding floating point values.

force_ascii [boolean, default True] Force encoded string to be ASCII.

date_unit [string, default ‘ms’ (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’, ‘ns’ for second, millisecond, microsecond, and nanosecond respectively.

default_handler [callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines [boolean, default False] If ‘orient’ is ‘records’ write out line delimited json format. Will throw ValueError if incorrect ‘orient’ since others are not list like.

New in version 0.19.0.

compression [{None, ‘gzip’, ‘bz2’, ‘zip’, ‘xz’}] A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

index [boolean, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is ‘split’ or ‘table’.

New in version 0.23.0.

pandas.read_json

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]}'
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

to_latex(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=False*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires \usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

to_latex-specific options:

bold_rows [boolean, default False] Make the row labels bold in the output

column_format [str, default None] The columns format as specified in LaTeX table format e.g 'rcl' for 3 columns

longtable [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a \usepackage{longtable} to your LaTeX preamble.

escape [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

encoding [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

multicolumn [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module.

New in version 0.20.0.

multirow [boolean, default False] Use multirow to enhance MultiIndex rows. Requires adding a \usepackage{multirow} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

New in version 0.20.0.

to_msgpack (*path_or_buf=None*, *encoding='utf-8'*, ***kwargs*)
msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

path [string File path, buffer-like, or None] if None, return generated string

append [boolean whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

to_panel ()
Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Deprecated since version 0.20.0.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

panel : Panel

to_parquet (*fname*, *engine='auto'*, *compression='snappy'*, ***kwargs*)
Write a DataFrame to the binary parquet format.

New in version 0.21.0.

This function writes the dataframe as a parquet file. You can choose different parquet backends, and have the option of compression. See the user guide for more details.

fname [str] String file path.

engine [{'auto', 'pyarrow', 'fastparquet'}, default 'auto'] Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

compression [{'snappy', 'gzip', 'brotli', None}, default 'snappy'] Name of the compression to use. Use `None` for no compression.

****kwargs** Additional arguments passed to the parquet library. See pandas io for more details.

read_parquet : Read a parquet file. DataFrame.to_csv : Write a csv file. DataFrame.to_sql : Write to a sql table. DataFrame.to_hdf : Write to hdf.

This function requires either the fastparquet or pyarrow library.

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gzip', compression='gzip')
>>> pd.read_parquet('df.parquet.gzip')
   col1  col2
```

(continues on next page)

(continued from previous page)

```
0      1      3
1      2      4
```

to_period (*freq=None*, *axis=0*, *copy=True*)

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

freq : string, default axis : {0 or 'index', 1 or 'columns'}, default 0

The axis to convert (the index by default)

copy [boolean, default True] If False then underlying input data is not copied

ts : TimeSeries with PeriodIndex

to_pickle (*path*, *compression='infer'*, *protocol=2*)

Pickle (serialize) object to file.

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

protocol [int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1]_ paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

read_pickle : Load pickled pandas object (or any object) from file. DataFrame.to_hdf : Write DataFrame to an HDF5 file. DataFrame.to_sql : Write DataFrame to a SQL database. DataFrame.to_parquet : Write a DataFrame to the binary parquet format.

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

to_records(*index=True*, *convert_datetime64=None*)

Convert DataFrame to a NumPy record array.

Index will be put in the 'index' field of the record array if requested.

index [boolean, default True] Include index in resulting record array, stored in 'index' field.

convert_datetime64 [boolean, default None] Deprecated since version 0.23.0.

Whether to convert the index to datetime.datetime if it is a DatetimeIndex.

y : numpy.recarray

DataFrame.from_records: convert structured or record ndarray to DataFrame.

numpy.recarray: ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   A     B
a  1  0.50
b  2  0.75
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

By default, timestamps are converted to *datetime.datetime*:

```
>>> df.index = pd.date_range('2018-01-01 09:00', periods=2, freq='min')
>>> df
                     A     B
2018-01-01 09:00:00  1  0.50
2018-01-01 09:01:00  2  0.75
>>> df.to_records()
rec.array([(datetime.datetime(2018, 1, 1, 9, 0), 1, 0.5 ),
           (datetime.datetime(2018, 1, 1, 9, 1), 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The timestamp conversion can be disabled so NumPy's datetime64 data type is used instead:

```
>>> df.to_records(convert_datetime64=False)
rec.array([('2018-01-01T09:00:00.000000000', 1, 0.5 ),
           ('2018-01-01T09:01:00.000000000', 2, 0.75)],
          dtype=[('index', '<M8[ns]'), ('A', '<i8'), ('B', '<f8')])
```

to_sparse(*fill_value=None*, *kind='block'*)

Convert to SparseDataFrame

fill_value : float, default NaN kind : {'block', 'integer'}

y : SparseDataFrame

to_sql(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1]_ are supported. Tables can be newly created, appended to, or overwritten.

name [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists [{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index [boolean, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize [int, optional] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

ValueError When the table already exists and *if_exists* is 'fail' (the default).

pandas.read_sql : read a DataFrame from a table

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
```

(continues on next page)

(continued from previous page)

```
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

to_stata(*fname*, *convert_dates=None*, *write_index=True*, *encoding='latin-1'*, *byteorder=None*, *time_stamp=None*, *data_label=None*, *variable_labels=None*, *version=114*, *convert_strl=None*)

Export Stata binary dta files.

fname [path (string), buffer or path object] string, path object (pathlib.Path or py._path.local.LocalPath) or object implementing a binary write() functions. If using a buffer then the buffer will not be automatically closed after the file data has been written.

convert_dates [dict] Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to 'tc'. Raises NotImplementedError if a datetime column has timezone information.

write_index [bool] Write the index to Stata dataset.

encoding [str] Default is latin-1. Unicode is not supported.

byteorder [str] Can be ">", "<", "little", or "big". default is *sys.byteorder*.

time_stamp [datetime] A datetime to use as file creation date. Default is the current time.

data_label [str] A label for the data set. Must be 80 characters or smaller.

variable_labels [dict] Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

New in version 0.19.0.

version [{114, 117}] Version to use in the output dta file. Version 114 can be used read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 114 limits string variables to 244 characters

or fewer while 117 allows strings with lengths up to 2,000,000 characters.

New in version 0.23.0.

convert_strl [list, optional] List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

New in version 0.23.0.

NotImplementedError

- If datetimes contain timezone information
- Column dtype is not representable in Stata

ValueError

- Columns listed in convert_dates are neither datetime64[ns] or datetime.datetime
- Column listed in convert_dates is not in DataFrame
- Categorical label contains more than 32,000 characters

New in version 0.19.0.

pandas.read_stata : Import Stata data files pandas.io.stata.StataWriter : low-level writer for Stata data files pandas.io.stata.StataWriter117 : low-level writer for version 117 files

```
>>> data.to_stata('./data_file.dta')
```

Or with dates

```
>>> data.to_stata('./date_data_file.dta', {2 : 'tw'})
```

Alternatively you can create an instance of the StataWriter class

```
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

With dates:

```
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
```

to_string(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *line_width=None*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*)

Render a DataFrame to a console-friendly tabular output.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_timestamp (*freq=None*, *how='start'*, *axis=0*, *copy=True*)
Cast to DatetimeIndex of timestamps, at *beginning* of period

freq [string, default frequency of PeriodIndex] Desired frequency

how [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default)

copy [boolean, default True] If false then underlying input data is not copied

df : DataFrame with DatetimeIndex

to_xarray()
Return an xarray object from the pandas object.

a DataArray for a Series a Dataset for a DataFrame a DataArray for higher dims

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
```

(continues on next page)

(continued from previous page)

```
   A    B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index    (index) int64 0 1 2
Data variables:
    A        (index) int64 1 1 2
    B        (index) object 'foo' 'bar' 'foo'
    C        (index) float64 4.0 5.0 6.0
```

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)}
                      ).set_index(['B','A'])
>>> df
         C
B   A
foo 1  4.0
bar 1  5.0
foo 2  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B        (B) object 'bar' 'foo'
  * A        (A) int64 1 2
Data variables:
    C        (B, A) float64 5.0 nan 4.0 6.0
```

```
>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                 items=list('ABCD'),
                 major_axis=pd.date_range('20130101', periods=3),
                 minor_axis=['first', 'second'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second
```

```
>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],
       [[ 6,  7],
        [ 8,  9],
        [10, 11]],
       [[12, 13],
```

(continues on next page)

(continued from previous page)

```
        [14, 15],
        [16, 17]],
       [[18, 19],
        [20, 21],
        [22, 23]]])
Coordinates:
  * items       (items) object 'A' 'B' 'C' 'D'
  * major_axis  (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03
↪# noqa
  * minor_axis  (minor_axis) object 'first' 'second'
```

See the xarray docs

transform(*func*, **args*, ***kwargs*)

Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values

New in version 0.20.0.

func [callable, string, dictionary, or list of string/callables] To apply to column

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

transformed : NDFrame

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
...                   index=pd.date_range('1/1/2000', periods=10))
df.iloc[3:7] = np.nan
```

```
>>> df.transform(lambda x: (x - x.mean()) / x.std())
                   A         B         C
2000-01-01  0.579457  1.236184  0.123424
2000-01-02  0.370357 -0.605875 -1.231325
2000-01-03  1.455756 -0.277446  0.288967
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -0.498658  1.274522  1.642524
2000-01-09 -0.540524 -1.012676 -0.828968
2000-01-10 -1.366388 -0.614710  0.005378
```

pandas.NDFrame.aggregate pandas.NDFrame.apply

transpose(**args*, ***kwargs*)

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
                0     1
name        Alice   Bob
score         9.5     8
employed    False  True
kids            0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

truediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

before [date, string, int] Truncate all rows before this index value.

after [date, string, int] Truncate all rows after this index value.

axis [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

copy [boolean, default is True,] Return a copy of the truncated section.

type of caller The truncated Series or DataFrame.

DataFrame.loc : Select a subset of a DataFrame by label. DataFrame.iloc : Select a subset of a DataFrame by position.

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
```

(continues on next page)

(continued from previous page)

```
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

tshift (*periods=1*, *freq=None*, *axis=0*)
Shift the time index, using the index's frequency if available.

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

shifted : NDFrame

tz_convert (*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

tz : string or pytz.timezone object axis : the axis to convert level : int, str, default None

If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

TypeError If the axis is tz-naive.

tz_localize (*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*)
Localize tz-naive TimeSeries to target time zone.

tz : string or pytz.timezone object axis : the axis to localize level : int, str, default None

If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise']

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times

- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

TypeError If the TimeSeries is tz-aware and tz is not None.

unstack (*level=-1*, *fill_value=None*)

Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels. If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex). The level involved will automatically get sorted.

level [int, string, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name

fill_value [replace NaN with this value if the unstack produces] missing values

New in version 0.18.0.

DataFrame.pivot : Pivot a table based on column values. DataFrame.stack : Pivot a level of the column labels (inverse operation

from *unstack*).

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                    ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a   1.0
     b   2.0
two  a   3.0
     b   4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a   b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
   one  two
a  1.0   3.0
b  2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a  1.0
     b  2.0
two  a  3.0
     b  4.0
dtype: float64
```

unstacked : DataFrame or Series

update (*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

other [DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

join [{'left'}, default 'left'] Only left join is implemented, keeping the index and columns of the original object.

overwrite [bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.
- False: only update values that are NA in the original DataFrame.

filter_func [callable(1d-array) -> boolean 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

raise_conflict [bool, default False] If True, will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

ValueError When *raise_conflict* is True and there's overlapping non-NA data.

dict.update : Similar method for dictionaries. DataFrame.merge : For column(s)-on-columns(s) operations.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
1  b  e
2  c  f
```

For Series, it's name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
```

(continues on next page)

(continued from previous page)

```
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A      B
0  1    4.0
1  2  500.0
2  3    6.0
```

values

Return a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

numpy.ndarray The values of the DataFrame.

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age':    [ 3,  29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3      94      31
1   29     170     115
>>> df.dtypes
age       int64
height    int64
weight    int64
dtype: object
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('monkey', np.nan, None)],
...                    columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name          object
max_speed    float64
rank          object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
```

(continues on next page)

(continued from previous page)

```
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

pandas.DataFrame.index : Retrievie the index labels pandas.DataFrame.columns : Retrieving the column names

var(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

var : Series or DataFrame (if level specified)

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
```

(continues on next page)

(continued from previous page)

```
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

DataFrame.mask()

xs(*key*, *axis=0*, *level=None*, *drop_level=True*)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on the rows (axis=0).

key [object] Some label contained in the index, or partially in a MultiIndex

axis [int, default 0] Axis to retrieve cross-section on

level [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level [boolean, default True] If False, returns object with same levels as self.

```
>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
A    4
B    5
C    2
Name: a
>>> df.xs('C', axis=1)
a    2
b    9
c    3
Name: C
```

```
>>> df
                     A  B  C  D
first second third
bar   one    1       4  1  8  9
      two    1       7  5  5  0
baz   one    1       6  6  8  0
      three  2       5  3  5  3
>>> df.xs(('baz', 'three'))
       A  B  C  D
third
2      5  3  5  3
>>> df.xs('one', level=1)
             A  B  C  D
first third
bar   1      4  1  8  9
baz   1      6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
        A  B  C  D
```

(continues on next page)

(continued from previous page)

```
second
three   5  3  5  3
```

xs : Series or DataFrame

xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of xs functionality, see MultiIndex Slicers

class Fred2.Core.Result.**EpitopePredictionResult**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

Bases: *Fred2.Core.Result.AResult*

A *EpitopePredictionResult* object is a DataFrame with multi-indexing, where column Ids are the prediction model (i.e HLA *Allele* for epitope prediction), row ID the target of the prediction (i.e. *Peptide*) and the second row ID the predictor (e.g. BIMAS)

EpitopePredictionResult

Peptide Obj	Method Name	Allele1 Obj	Allele2 Obj	Allele3 Obj
Peptide1	Method 1	0.324	0.56	0.013
	Method 2	20	15	23
Peptide2	Method 1	0.50	0.36	0.98
	Method 2	26	10	50

T

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
```

(continues on next page)

(continued from previous page)

```
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
              0     1
name      Alice   Bob
score       9.5     8
employed  False  True
kids          0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

abs()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

abs Series/DataFrame containing the absolute value of each element.

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
     a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
     a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

numpy.absolute : calculate the absolute value element-wise.

add(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.radd

add_prefix(*prefix*)

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

prefix [str] The string to add before each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_suffix: Suffix row labels with string *suffix*. DataFrame.add_suffix: Suffix column labels with string *suffix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0       1       3
1       2       4
2       3       5
3       4       6
```

add_suffix(*suffix*)

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

suffix [str] The string to add after each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_prefix: Prefix row labels with string *prefix*. DataFrame.add_prefix: Prefix column labels with string *prefix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

agg (*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

aggregate (*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

align(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)

Align two objects on their axes with the specified join method for each axis Index

other : DataFrame or Series join : {'outer', 'inner', 'left', 'right'}, default 'outer' axis : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None)

level [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level

copy [boolean, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

method : str, default None limit : int, default None fill_axis : {0 or 'index', 1 or 'columns'}, default 0

Filling axis, method and limit

broadcast_axis [{0 or 'index', 1 or 'columns'}, default None] Broadcast values along this axis, if aligning two objects of different dimensions

(left, right) [(DataFrame, type of other)] Aligned objects

all (*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether all elements are True, potentially over an axis.

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

all : Series or DataFrame (if level specified)

pandas.Series.all : Return True if all elements are True pandas.DataFrame.any : Return True if one (or more) elements are True

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

any(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

any : Series or DataFrame (if level specified)

pandas.DataFrame.all : Return whether all elements are True.

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([True, False]).any()
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
```

(continues on next page)

(continued from previous page)

```
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

append(*other*, *ignore_index=False*, *verify_integrity=False*, *sort=None*)

Append rows of *other* to the end of this frame, returning a new object. Columns not in this frame are added as new columns.

other [DataFrame or Series/dict-like object, or list of these] The data to append.

ignore_index [boolean, default False] If True, do not use the index labels.

verify_integrity [boolean, default False] If True, raise ValueError on creating index with duplicates.

sort [boolean, default None] Sort columns if the columns of *self* and *other* are not aligned. The default sorting is deprecated and will change to not-sorting in a future version of pandas. Explicitly pass `sort=True` to silence the warning and sort. Explicitly pass `sort=False` to silence the warning and not sort.

New in version 0.23.0.

appended : DataFrame

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

pandas.concat [General function to concatenate DataFrame, Series] or Panel objects

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With *ignore_index* set to True:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
...     df = df.append({'A': i}, ignore_index=True)
>>> df
   A
0  0
1  1
2  2
3  3
4  4
```

More efficient:

```
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
...           ignore_index=True)
   A
0  0
1  1
2  2
3  3
4  4
```

apply (*func*, *axis=0*, *broadcast=None*, *raw=False*, *reduce=None*, *result_type=None*, *args=()*, ***kwds*)
Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

func [function] Function to apply to each column or row.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

broadcast [bool, optional] Only relevant for aggregation functions:

- `False` or `None` : returns a Series whose length is the length of the index or the number of columns (based on the *axis* parameter)
- `True` : results will be broadcast to the original shape of the frame, the original index and columns will be retained.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by result_type='broadcast'.

raw [bool, default False]

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

reduce [bool or None, default None] Try to apply reduction procedures. If the DataFrame is empty, *apply* will use *reduce* to determine whether the result should be a Series or a DataFrame. If `reduce=None` (the default), *apply*'s return value will be guessed by calling *func* on an empty Series (note: while guessing, exceptions raised by *func* will be ignored). If `reduce=True` a Series will always be returned, and if `reduce=False` a DataFrame will always be returned.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by `result_type='reduce'`.

result_type [{'expand', 'reduce', 'broadcast', None}, default None] These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.
- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

New in version 0.23.0.

args [tuple] Positional arguments to pass to *func* in addition to the array/series.

****kwds** Additional keyword arguments to pass as keywords arguments to *func*.

In the current implementation apply calls *func* twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if *func* has side-effects, as they will take effect twice for the first column/row.

DataFrame.applymap: For elementwise operations DataFrame.aggregate: only perform aggregating type operations DataFrame.transform: only perform transformating type operations

```
>>> df = pd.DataFrame([[4, 9],] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
```

(continues on next page)

(continued from previous page)

```
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Retuning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing result_type='expand' will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
```

(continues on next page)

(continued from previous page)

```
1  1  2
2  1  2
```

applied : Series or DataFrame

applymap (*func*)

Apply a function to a Dataframe elementwise.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

func [callable] Python function, returns a single value from a single value.

DataFrame Transformed DataFrame.

DataFrame.apply : Apply a function along input axis of DataFrame

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.applymap(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.applymap(lambda x: x**2)
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

But it's better to avoid applymap in that case.

```
>>> df ** 2
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

as_blocks (*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

copy : boolean, default True

values : a dict of dtype -> Constructor Types

as_matrix (*columns=None*)

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

columns: list, optional, default:None If None, return all columns, otherwise, returns specified columns.

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a flot64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

pandas.DataFrame.values

asfreq(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)
Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

freq : DateOffset object, or string method : {'backfill'/'bfill', 'pad'/'ffill'}, default None

Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill

how [{'start', 'end'}, default end] For PeriodIndex only, see PeriodIndex.asfreq

normalize [bool, default False] Whether to reset output index to midnight

fill_value: scalar, optional Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

converted : type of caller

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:03:00    3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                     s
2000-01-01 00:00:00    0.0
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    2.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    3.0
2000-01-01 00:03:00    3.0
```

reindex

To learn more about the frequency strings, please see this link.

asof (*where*, *subset=None*)

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

where : date or array of dates subset : string or list of strings, default None

if not None use these columns for NaN propagation

Dates are assumed to be sorted Raises if this is not the case

where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

where is Index: same shape object as input

merge_asof

assign (***kwargs*)

Assign new columns to a DataFrame, returning a new object (a copy) with the new columns added to the original ones. Existing columns that are re-assigned will be overwritten.

kwargs [keyword, value pairs] keywords are the column names. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

df [DataFrame] A new DataFrame with the new columns in addition to all the existing columns.

Assigning multiple columns within the same `assign` is possible. For Python 3.6 and above, later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order. For Python 3.5 and below, the order of keyword arguments is not specified, you cannot refer to newly created or modified columns. All items are computed first, and then assigned in alphabetical order.

Changed in version 0.23.0: Keyword argument order is maintained for Python 3.6 and later.

```
>>> df = pd.DataFrame({'A': range(1, 11), 'B': np.random.randn(10)})
```

Where the value is a callable, evaluated on *df*:

```
>>> df.assign(ln_A = lambda x: np.log(x.A))
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the value already exists and is inserted:

```
>>> newcol = np.log(df['A'])
>>> df.assign(ln_A=newcol)
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the keyword arguments depend on each other

```
>>> df = pd.DataFrame({'A': [1, 2, 3]})
```

```
>>> df.assign(B=df.A, C=lambda x:x['A']+ x['B'])
   A  B  C
 0  1  1  2
 1  2  2  4
 2  3  3  6
```

astype(***kwargs*)

Cast a pandas object to a specified dtype `dtype`.

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy [bool, default True.] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors [{'raise', 'ignore'}, default 'raise'.] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

raise_on_error [raise on invalid input] Deprecated since version 0.20.0: Use `errors` instead

kwargs : keyword arguments to pass on to the constructor

casted : type of caller

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> ser.astype('category', ordered=True, categories=[2, 1])
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to a numeric type. numpy.ndarray.astype : Cast a numpy array to a specified type.

at

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.iat [Access a single value for a row/column pair by integer] position

DataFrame.loc : Access a group of rows and columns by label(s) Series.at : Access a single value using a label

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

KeyError When label does not exist in DataFrame

at_time (*time*, *asof=False*)

Select values at particular time of day (e.g. 9:30AM).

TypeError If the index is not a `DatetimeIndex`

time : datetime.time or string

values_at_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

between_time : Select values between particular times of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_at_time : Get just the index locations for

values at particular time of the day

axes

Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['col1', 'col2'],
dtype='object')]
```

between_time (*start_time*, *end_time*, *include_start=True*, *include_end=True*)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

TypeError If the index is not a `DatetimeIndex`

start_time : datetime.time or string end_time : datetime.time or string include_start : boolean, default True include_end : boolean, default True

values_between_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

at_time : Select values at a particular time of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_between_time : Get just the index locations for

values between particular times of the day

bfill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='bfill')`

blocks
Internal property, property synonym for as_blocks()

Deprecated since version 0.21.0.

bool()
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

boxplot(*column=None*, *by=None*, *ax=None*, *fontsize=None*, *rot=0*, *grid=True*, *figsize=None*, *layout=None*, *return_type=None*, ***kwds*)
Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to *1.5 * IQR (IQR = Q3 - Q1)* from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia's entry for boxplot.

column [str or list of str, optional] Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.

by [str or array-like, optional] Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in *by*.

ax [object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.

fontsize [float or str] Tick label font size in points or as a string (e.g., *large*).

rot [int or float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate sytem.

grid [boolean, default True] Setting this to True will show the grid.

figsize [A tuple (width, height) in inches] The size of the figure to create in matplotlib.

layout [tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 columns and 5 rows, starting from the top-left.

return_type [{'axes', 'dict', 'both'} or None, default 'axes'] The kind of object to return. The default is `axes`.

- 'axes' returns the matplotlib axes the boxplot is drawn on.
- 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot.
- 'both' returns a namedtuple with the axes and dict.
- when grouping with `by`, a Series mapping columns to `return_type` is returned.

  If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

result :

The return type depends on the *return_type* parameter:

- 'axes' : object of class matplotlib.axes.Axes

- 'dict' : dict of matplotlib.lines.Line2D objects
- 'both' : a nametuple with strucure (ax, lines)

For data grouped with `by`:

- `Series`
- `array` (for `return_type = None`)

Series.plot.hist: Make a histogram. matplotlib.pyplot.boxplot : Matplotlib equivalent plot.

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`. For instance:

A list of strings (i.e. `['X', 'Y']`) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

The layout of boxplot can be adjusted giving a tuple to `layout`:

Additional formatting can be done to the boxplot, like suppressing the grid (`grid=False`), rotating the labels in the x-axis (i.e. `rot=45`) or changing the fontsize (i.e. `fontsize=15`):

The parameter `return_type` can be used to select the type of element returned by *boxplot*. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```
>>> boxplot = df.boxplot(column=['Col1','Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

When grouping with `by`, a Series mapping columns to `return_type` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned:

```
>>> boxplot =  df.boxplot(column=['Col1', 'Col2'], by='X',
...                       return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

`clip` (*lower=None*, *upper=None*, *axis=None*, *inplace=False*, **args*, ***kwargs*)
Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

lower [float or array_like, default None] Minimum threshold value. All values below this threshold will be set to it.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

clip_lower : Clip values below specified threshold(s). clip_upper : Clip values above specified threshold(s).

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

clip_lower(*threshold*, *axis=None*, *inplace=False*)

Return copy of the input with values below a threshold truncated.

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Series.clip [Return copy of input with values below and above] thresholds truncated.

Series.clip_upper [Return copy of input with values above] threshold truncated.

clipped : same type as input

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

clip_upper (*threshold*, *axis=None*, *inplace=False*)

Return copy of input with values above given value(s) truncated.

threshold : float or array_like axis : int or string axis name, optional

Align object with threshold along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data

New in version 0.21.0.

clip

clipped : same type as input

columns

The column labels of the DataFrame.

combine (*other*, *func*, *fill_value=None*, *overwrite=True*)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

other : DataFrame func : function

Function that takes two series as inputs and return a Series or a scalar

fill_value : scalar value overwrite : boolean, default True

If True then overwrite values for common keys in the calling frame

result : DataFrame

```
>>> df1 = DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, lambda s1, s2: s1 if s1.sum() < s2.sum() else s2)
   A  B
0  0  3
1  0  3
```

DataFrame.combine_first [Combine two DataFrame objects and default to] non-null values in frame calling the method

combine_first (*other*)

Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

other : DataFrame

combined : DataFrame

df1's values prioritized, use values from df2 to fill holes:

```
>>> df1 = pd.DataFrame([[1, np.nan]])
>>> df2 = pd.DataFrame([[3, 4]])
>>> df1.combine_first(df2)
   0    1
0  1  4.0
```

DataFrame.combine [Perform series-wise operation on two DataFrames] using a given function

compound(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

compounded : Series or DataFrame (if level specified)

consolidate(*inplace=False*)

Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

convert_objects(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Return a fixed frequency timedelta index,

with day as the default.

converted : same as input object

copy(*deep=True*)

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

deep [bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

copy [Series, DataFrame or Panel] Object type matches caller.

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
```

(continues on next page)

(continued from previous page)

```
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

corr (*method='pearson'*, *min_periods=1*)

Compute pairwise correlation of columns, excluding NA/null values

method [{'pearson', 'kendall', 'spearman'}]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation

y : DataFrame

corrwith (*other*, *axis=0*, *drop=False*)

Compute pairwise correlation between rows or columns of two DataFrame objects.

other : DataFrame, Series axis : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' to compute column-wise, 1 or 'columns' for row-wise

drop [boolean, default False] Drop missing indices from result, default returns union of all

correls : Series

count (*axis=0*, *level=None*, *numeric_only=False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, and optionally *numpy.inf* (depending on *pandas.options.mode.use_inf_as_na*) are considered NA.

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each **row**.

level [int or str, optional] If the axis is a *MultiIndex* (hierarchical), count along a particular *level*, collapsing into a *DataFrame*. A *str* specifies the level name.

numeric_only [boolean, default False] Include only *float*, *int* or *boolean* data.

Series or DataFrame For each column/row the number of non-NA/null entries. If *level* is specified returns a *DataFrame*.

Series.count: number of non-NA elements in a Series DataFrame.shape: number of DataFrame rows and columns (including NA

elements)

DataFrame.isna: boolean same-sized DataFrame showing places of NA elements

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                    ["John", "Myla", None, "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> df
   Person   Age  Single
0    John  24.0   False
1    Myla   NaN    True
2    None  21.0    True
3    John  33.0    True
4    Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    4
Age       4
Single    5
dtype: int64
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    2
3    3
4    3
dtype: int64
```

Counts for one level of a *MultiIndex*:

```
>>> df.set_index(["Person", "Single"]).count(level="Person")
        Age
Person
John      2
Myla      1
```

cov (*min_periods=None*)

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as `NaN`.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result.

DataFrame The covariance matrix of the series of the DataFrame.

pandas.Series.cov : compute covariance with another Series pandas.core.window.EWM.cov: expoential weighted sample covariance pandas.core.window.Expanding.cov : expanding sample covariance pandas.core.window.Rolling.cov : rolling sample covariance

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                   columns=['dogs', 'cats'])
>>> df.cov()
          dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
          a         b         c         d         e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
```

(continues on next page)

(continued from previous page)

```
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
          a         b         c
a  0.316741       NaN -0.150812
b       NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

cummax (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummax : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.max [Similar functionality] but ignores `NaN` values.

DataFrame.max [Return the maximum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cummin (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummin : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
```

(continues on next page)

(continued from previous page)

```
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.min [Similar functionality] but ignores `NaN` values.

DataFrame.min [Return the minimum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumprod (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumprod : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.prod [Similar functionality] but ignores `NaN` values.

DataFrame.prod [Return the product over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumsum(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumsum : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.sum [Similar functionality] but ignores `NaN` values.

DataFrame.sum [Return the sum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

`describe` (*percentiles=None*, *include=None*, *exclude=None*)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

summary: Series/DataFrame of summary statistics

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                       3
unique                      2
top       2010-01-01 00:00:00
freq                        2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                     'numeric': [1, 2, 3],
...                     'categorical': pd.Categorical(['d','e','f'])
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
        categorical  numeric object
count             3      3.0      3
unique            3      NaN      3
top               f      NaN      c
freq              1      NaN      1
mean            NaN      2.0    NaN
std             NaN      1.0    NaN
min             NaN      1.0    NaN
25%             NaN      1.5    NaN
50%             NaN      2.0    NaN
75%             NaN      2.5    NaN
max             NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
        categorical  numeric
count            3      3.0
```

(continues on next page)

(continued from previous page)

```
unique            3      NaN
top               f      NaN
freq              1      NaN
mean            NaN      2.0
std             NaN      1.0
min             NaN      1.0
25%             NaN      1.5
50%             NaN      2.0
75%             NaN      2.5
max             NaN      3.0
```

DataFrame.count DataFrame.max DataFrame.min DataFrame.mean DataFrame.std DataFrame.select_dtypes

diff (*periods=1*, *axis=0*)

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

periods [int, default 1] Periods to shift for calculating difference, accepts negative values.

axis [{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

New in version 0.16.1..

diffed : DataFrame

Series.diff: First discrete difference for a Series. DataFrame.pct_change: Percent change over given number of periods. DataFrame.shift: Shift index by desired number of periods with an

optional time freq.

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a    b     c
0 NaN  0.0   0.0
1 NaN -1.0   3.0
2 NaN -1.0   7.0
3 NaN -1.0  13.0
4 NaN  0.0  20.0
5 NaN  2.0  28.0
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

div(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

divide(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

dot(*other*)

Matrix multiplication with DataFrame or Series objects. Can also be called using *self @ other* in Python >= 3.5.

other : DataFrame or Series

dot_product : DataFrame or Series

drop(*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

labels [single label or list-like] Index or column labels to drop.

axis [{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index, columns [single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

New in version 0.21.0.

level [int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace [bool, default False] If True, do operation inplace and return None.

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

dropped : pandas.DataFrame

DataFrame.loc : Label-location based indexer for selection by label. DataFrame.dropna : Return DataFrame with labels on given axis omitted

where (all or any) data are missing

DataFrame.drop_duplicates [Return DataFrame with duplicate rows] removed, optionally only considering certain columns

Series.drop : Return Series with specified index labels removed.

KeyError If none of the labels are found in the selected axis

```
>>> df = pd.DataFrame(np.arange(12).reshape(3,4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                              [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3,0.2]])
>>> df
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
lama    speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
```

(continues on next page)

(continued from previous page)

```
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

drop_duplicates (*subset=None*, *keep='first'*, *inplace=False*)

Return DataFrame with duplicate rows removed, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Drop duplicates except for the first occurrence.
- `last` : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

inplace [boolean, default False] Whether to drop duplicates in place or to return a copy

deduplicated : DataFrame

dropna (*axis=0*, *how='any'*, *thresh=None*, *subset=None*, *inplace=False*)

Remove missing values.

See the User Guide for more on which values are considered missing, and how to work with missing data.

axis [{0 or 'index', 1 or 'columns'}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Deprecated since version 0.23.0:: Pass tuple or list to drop on multiple

axes.

how [{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

thresh [int, optional] Require that many non-NA values.

subset [array-like, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace [bool, default False] If True, do operation inplace and return None.

DataFrame DataFrame with NA entries dropped from it.

DataFrame.isna: Indicate missing values. DataFrame.notna : Indicate existing (non-missing) values. DataFrame.fillna : Replace missing values. Series.dropna : Drop missing values. Index.dropna : Drop missing indices.

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'born'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type of each column.

pandas.DataFrame.ftypes : dtype and sparsity information.

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
string              object
dtype: object
```

duplicated(*subset=None*, *keep='first'*)

Return boolean Series denoting duplicate rows, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Mark duplicates as `True` except for the first occurrence.
- `last` : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

duplicated : Series

empty

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

bool If DataFrame is empty, return True, if not return False.

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
```

(continues on next page)

(continued from previous page)

```
>>> df.dropna().empty
True
```

pandas.Series.dropna pandas.DataFrame.dropna

eq(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods eq

equals(*other*)
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

eval(*expr*, *inplace=False*, ***kwargs*)
Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

expr [str] The expression string to evaluate.

inplace [bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

New in version 0.18.0..

kwargs [dict] See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

ndarray, scalar, or pandas object The result of the evaluation.

DataFrame.query [Evaluates a boolean expression to query the columns] of a frame.

DataFrame.assign [Can evaluate an expression or function to create new] values for a column.

pandas.eval [Evaluate a Python expression as a string using various] backends.

For more details see the API documentation for `eval()`. For detailed examples see enhancing performance with eval.

```
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
>>> df.eval('A + B')
0    11
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
```

Use `inplace=True` to modify the original DataFrame.

```
>>> df.eval('C = A + B', inplace=True)
>>> df
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
```

ewm(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*)
Provides exponential weighted functions

New in version 0.18.0.

com [float, optional] Specify decay in terms of center of mass, $\alpha = 1/(1 + com),$ for $com \geq 0$

span [float, optional] Specify decay in terms of span, $\alpha = 2/(span + 1),$ for $span \geq 1$

halflife [float, optional] Specify decay in terms of half-life, $\alpha = 1 - exp(log(0.5)/halflife),$ for $halflife > 0$

alpha [float, optional] Specify smoothing factor α directly, $0 < \alpha \leq 1$

New in version 0.18.0.

min_periods [int, default 0] Minimum number of observations in window required to have a value (otherwise result is NA).

adjust [boolean, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

ignore_na [boolean, default False] Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), . . . , 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

rolling : Provides rolling window calculations expanding : Provides expanding transformations.

expanding (*min_periods=1*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

min_periods [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

center [boolean, default False] Set the labels at the center of the window.

axis : int or string, default 0

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

rolling : Provides rolling window calculations ewm : Provides exponential weighted functions

ffill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='ffill')`

fillna(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)
Fill NA/NaN values using the specified method

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis : {0 or 'index', 1 or 'columns'} inplace : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

interpolate : Fill NaN values using interpolation. reindex, asfreq

filled : DataFrame

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 2.0 1
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

filter (*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

items [list-like] List of info axis to restrict to (must not all be present)

like [string] Keep info axis where "arg in col == True"

regex [string (regular expression)] Keep info axis with re.search(regex, col) == True

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame

same type as input object

```
>>> df
one  two  three
mouse     1    2      3
rabbit    4    5      6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
one  two  three
rabbit    4    5      6
```

pandas.DataFrame.loc

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

filter_result(*expressions*)

Filters a result data frame based on a specified expression consisting of a list of triple with (method_name, comparator, threshold). The expression is applied to each row. If any of the columns fulfill the criteria the row remains.

Parameters **expressions** (*list((str, comparator, float))*) – A list of triples consisting of (method_name, comparator, threshold)

Returns Filtered result object

Return type *EpitopePredictionResult*

first(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

subset : type of caller

last : Select final periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

first_valid_index()

Return index for first non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

floordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rfloordiv

classmethod from_csv(*path*, *header=0*, *sep=','*, *','*, *index_col=0*, *parse_dates=True*, *encoding=None*, *tupleize_cols=None*, *infer_datetime_format=False*)
Read CSV file.

Deprecated since version 0.21.0: Use `pandas.read_csv()` instead.

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a DataFrame of time series data.

This method only differs from the preferred `pandas.read_csv()` in some defaults:

- *index_col* is `0` instead of `None` (take first column as index by default)
- *parse_dates* is `True` instead of `False` (try parsing the index as datetime by default)

So a `pd.DataFrame.from_csv(path)` can be replaced by `pd.read_csv(path, index_col=0, parse_dates=True)`.

path : string file path or file handle / StringIO header : int, default 0

Row to use as header (skip prior rows)

sep [string, default ','] Field delimiter

index_col [int or sequence, default 0] Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

parse_dates [boolean, default True] Parse dates. Different default from read_table

tupleize_cols [boolean, default False] write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

infer_datetime_format: boolean, default False If True and *parse_dates* is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

pandas.read_csv

y : DataFrame

classmethod from_dict(*data*, *orient='columns'*, *dtype=None*, *columns=None*)
Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

data [dict] Of the form {field : array-like} or {field : dict}.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'.

dtype [dtype, default None] Data type to force, otherwise infer.

columns [list, default None] Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'`.

New in version 0.23.0.

pandas.DataFrame

DataFrame.from_records [DataFrame from ndarray (structured] dtype), list of tuples, dict, or DataFrame

DataFrame : DataFrame object creation using constructor

By default the keys of the dict become the DataFrame columns:

```
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Specify `orient='index'` to create the DataFrame using dictionary keys as rows:

```
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
       0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                        columns=['A', 'B', 'C', 'D'])
       A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
```

classmethod from_items(*items*, *columns=None*, *orient='columns'*)
Construct a dataframe from a list of tuples

Deprecated since version 0.23.0: *from_items* is deprecated and will be removed in a future version. Use `DataFrame.from_dict(dict(items))` instead. `DataFrame.from_dict(OrderedDict(items))` may be used to preserve the key order.

Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the columns, but depends on the specified orientation). The values should be arrays or Series.

items [sequence of (key, value) pairs] Values should be arrays or Series.

columns [sequence of column labels, optional] Must be passed if orient='index'.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the input correspond to column labels, pass 'columns' (default). Otherwise if the keys correspond to the index, pass 'index'.

frame : DataFrame

classmethod from_records(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float=False*, *nrows=None*)

Convert structured or record ndarray to DataFrame

data : ndarray (structured dtype), list of tuples, dict, or DataFrame index : string, list of fields, array-like

Field of array to use as the index, alternately a specific set of input labels to use

exclude [sequence, default None] Columns or fields to exclude

columns [sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

coerce_float [boolean, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

df : DataFrame

ftypes

Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type and indication of sparse/dense of each column.

pandas.DataFrame.dtypes: Series with just dtype information. pandas.SparseDataFrame : Container for sparse tabular data.

Sparse data should have the same dtypes as its dense representation.

```
>>> import numpy as np
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).ftypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).ftypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

ge(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods ge

get (*key*, *default=None*)
Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

key : object

value : type of items contained in object

get_dtype_counts ()
Return counts of unique dtypes in this object.

dtype [Series] Series with the count of columns with each dtype.

dtypes : Return the dtypes in this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

get_ftype_counts ()
Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

ftypes [Return ftypes (indication of sparse/dense and dtype) in] this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

get_value (*index*, *col*, *takeable=False*)
Quickly retrieve single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label takeable : interpret the index/col as indexers, default False

value : scalar value

get_values()
Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a *pandas.SparseArray*, the data are first converted to a dense representation.

numpy.ndarray Numpy representation of DataFrame

values : Numpy representation of DataFrame. pandas.SparseArray : Container for sparse data.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a      b    c
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
     a    c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

groupby(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=False*, *observed=False*, ***kwargs*)
Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

by [mapping, function, label, or list of labels] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted a (single) key.

axis : int, default 0 level : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

as_index [boolean, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort [boolean, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.

group_keys [boolean, default True] When calling apply, add group keys to index to identify pieces

squeeze [boolean, default False] reduce the dimensionality of the return type if possible, otherwise return a consistent type

observed [boolean, default False] This only applies if any of the groupers are Categoricals If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

GroupBy object

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

See the user guide for more.

resample [Convenience method for frequency conversion and resampling] of time series.

gt (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods gt

head (*n=5*)
Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

n [int, default 5] Number of rows to select.

obj_head [type of caller] The first *n* rows of the caller object.

pandas.DataFrame.tail: Returns the last *n* rows.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
```

(continues on next page)

(continued from previous page)

```
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

hist(*column=None*, *by=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *ax=None*, *sharex=False*, *sharey=False*, *figsize=None*, *layout=None*, *bins=10*, ***kwds*)
Make a histogram of the DataFrame's.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

data [DataFrame] The pandas object holding the data.

column [string or sequence] If passed, will be used to limit data to a subset of columns.

by [object, optional] If passed, then used to form histograms for separate groups.

grid [boolean, default True] Whether to show axis grid lines.

xlabelsize [int, default None] If specified changes the x-axis label size.

xrot [float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize [int, default None] If specified changes the y-axis label size.

yrot [float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

ax [Matplotlib axes object, default None] The axes to plot the histogram on.

sharex [boolean, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey [boolean, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize [tuple] The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

layout [tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

axes : matplotlib.AxesSubplot or numpy.ndarray of them

matplotlib.pyplot.hist : Plot a histogram using matplotlib.

iat

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.at : Access a single value for a row/column label pair DataFrame.loc : Access a group of rows and columns by label(s) DataFrame.iloc : Access a group of rows and columns by integer position(s)

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

IndexError When integer position is out of bounds

idxmax (*axis=0*, *skipna=True*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmax : Series

This method is the DataFrame version of `ndarray.argmax`.

Series.idxmax

idxmin (*axis=0*, *skipna=True*)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmin : Series

This method is the DataFrame version of `ndarray.argmin`.

Series.idxmin

iloc

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at Selection by Position

index

The index (row labels) of the DataFrame.

infer_objects()

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to numeric typeR

converted : same type as input object

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

info(*verbose=None*, *buf=None*, *max_cols=None*, *memory_usage=None*, *null_counts=None*)
Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

verbose [bool, optional] Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

buf [writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

max_cols [int, optional] When to switch from the verbose to the truncated output. If the DataFrame has more than *max_cols* columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

memory_usage [bool, str, optional] Specifies whether total memory usage of the DataFrame elements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources.

null_counts [bool, optional] Whether to show the non-null counts. By default, this is shown only if the frame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

None This method prints a summary of a DataFrame and returns None.

DataFrame.describe: Generate descriptive statistics of DataFrame columns.

DataFrame.memory_usage: Memory usage of DataFrame columns.

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                    "float_col": float_values})
>>> df
   int_col text_col  float_col
0        1    alpha       0.00
1        2     beta       0.25
2        3    gamma       0.50
3        4    delta       0.75
4        5  epsilon       1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
int_col      5 non-null int64
```

(continues on next page)

(continued from previous page)

```
text_col     5 non-null object
float_col    5 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w", encoding="utf-8") as f:
...     f.write(s)
260
```

The *memory_usage* parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 22.9+ MB
```

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 188.8 MB
```

insert (*loc*, *column*, *value*, *allow_duplicates=False*)

Insert column into DataFrame at specified location.

Raises a ValueError if *column* is already contained in the DataFrame, unless *allow_duplicates* is set to

True.

loc [int] Insertion index. Must verify 0 <= loc <= len(columns)

column [string, number, or hashable object] label of the inserted column

value : int, Series, or array-like allow_duplicates : bool, optional

interpolate(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

method [{'linear', 'time', 'index', 'values', 'nearest', 'zero',]

'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to `scipy.interpolate.interp1d`. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
- 'from_derivatives' refers to BPoly.from_derivatives which replaces 'piecewise_polynomial' interpolation method in scipy 0.18

New in version 0.18.1: Added support for the 'akima' method Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in scipy 0.18; backwards-compatible with scipy < 0.18

axis [{0, 1}, default 0]

- 0: fill column-by-column
- 1: fill row-by-row

limit [int, default None.] Maximum number of consecutive NaNs to fill. Must be greater than 0.

limit_direction : {'forward', 'backward', 'both'}, default 'forward' limit_area : {'inside', 'outside'}, default None

- None: (default) no fill restriction
- 'inside' Only fill NaNs surrounded by valid values (interpolate).
- 'outside' Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

inplace [bool, default False] Update the NDFrame in place if possible.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

kwargs : keyword arguments to pass on to the interpolating function.

Series or DataFrame of same shape interpolated at the NaNs

reindex, replace, fillna

Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

is_copy

isin(*values*)

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

values [iterable, Series, DataFrame or dictionary] The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dictionary, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

DataFrame of booleans

When `values` is a list:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
       A      B
0   True   True
1  False  False
2   True  False
```

When `values` is a dict:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
       A      B
0   True  False  # Note that B didn't match the 1 here.
1  False   True
2   True   True
```

When `values` is a Series or DataFrame:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
       A      B
0   True  False
1  False  False  # Column A in `other` has a 3, but not at index 1.
2   True   True
```

isna()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

isnull()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

items()

Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iteritems()

Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iterrows()

Iterate over DataFrame rows as (index, Series) pairs.

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
```

(continues on next page)

(continued from previous page)

```
>>> print(df['int'].dtype)
int64
```

To preserve dtypes while iterating over the rows, it is better to use *itertuples()* which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

it [generator] A generator that iterates over the rows of the frame.

itertuples : Iterate over DataFrame rows as namedtuples of the values. iteritems : Iterate over (column name, Series) pairs.

itertuples(*index=True*, *name='Pandas'*)

Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.

index [boolean, default True] If True, return the index as the first element of the tuple.

name [string, default "Pandas"] The name of the returned namedtuples or None to return regular tuples.

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. iteritems : Iterate over (column name, Series) pairs.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [0.1, 0.2]},
                      index=['a', 'b'])
>>> df
   col1  col2
a     1   0.1
b     2   0.2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='a', col1=1, col2=0.10000000000000001)
Pandas(Index='b', col1=2, col2=0.20000000000000001)
```

ix

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at Advanced Indexing.

join(*other*, *on=None*, *how='left'*, *lsuffix=''*, *rsuffix=''*, *sort=False*)

Join columns with other DataFrame either on index or on a key column. Efficiently Join multiple DataFrame objects by index at once by passing a list.

other [DataFrame, Series with name field set, or list of DataFrame] Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame

on [name, tuple/list of names, or array-like] Column or index level name(s) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

how [{'left', 'right', 'outer', 'inner'}, default: 'left'] How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use other frame's index
- outer: form union of calling frame's index (or column if on is specified) with other frame's index, and sort it lexicographically
- inner: form intersection of calling frame's index (or column if on is specified) with other frame's index, preserving the order of the calling's one

lsuffix [string] Suffix to use from left frame's overlapping columns

rsuffix [string] Suffix to use from right frame's overlapping columns

sort [boolean, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword)

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

Support for specifying index levels as the *on* parameter was added in version 0.23.0

```
>>> caller = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                        'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> caller
    A key
0  A0  K0
1  A1  K1
2  A2  K2
3  A3  K3
4  A4  K4
5  A5  K5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
    B key
0  B0  K0
1  B1  K1
2  B2  K2
```

Join DataFrames using their indexes.

```
>>> caller.join(other, lsuffix='_caller', rsuffix='_other')
```

```
>>>     A key_caller    B key_other
    0  A0         K0   B0        K0
    1  A1         K1   B1        K1
```

(continues on next page)

(continued from previous page)

```
    2  A2          K2   B2         K2
    3  A3          K3  NaN        NaN
    4  A4          K4  NaN        NaN
    5  A5          K5  NaN        NaN
```

If we want to join using the key columns, we need to set key to be the index in both caller and other. The joined DataFrame will have key as its index.

```
>>> caller.set_index('key').join(other.set_index('key'))
```

```
>>>      A    B
    key
    K0   A0   B0
    K1   A1   B1
    K2   A2   B2
    K3   A3  NaN
    K4   A4  NaN
    K5   A5  NaN
```

Another option to join using the key columns is to use the on parameter. DataFrame.join always uses other's index but we can use any column in the caller. This method preserves the original caller's index in the result.

```
>>> caller.join(other.set_index('key'), on='key')
```

```
>>>     A key    B
    0  A0  K0   B0
    1  A1  K1   B1
    2  A2  K2   B2
    3  A3  K3  NaN
    4  A4  K4  NaN
    5  A5  K5  NaN
```

DataFrame.merge : For column(s)-on-columns(s) operations

joined : DataFrame

keys()

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

kurt(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

kurtosis (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

last (*offset*)
Convenience method for subsetting final periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

subset : type of caller

first : Select initial periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

last_valid_index ()
Return index for last non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

le (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods le

loc
Access a group of rows and columns by label(s) or a boolean array.

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

> **Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at Selection by Label

DataFrame.at : Access a single value for a row/column label pair DataFrame.iloc : Access group of rows and columns by integer position(s) DataFrame.xs : Returns a cross-section (row(s) or column(s)) from the

Series/DataFrame.

Series.loc : Access group of values using labels

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=['cobra', 'viper', 'sidewinder'],
...       columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
```

(continues on next page)

(continued from previous page)

```
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

KeyError: when any items are not found

lookup(*row_labels*, *col_labels*)

Label-based "fancy indexing" function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

row_labels [sequence] The row labels to use for lookup

col_labels [sequence] The column labels to use for lookup

Akin to:

```
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

values [ndarray] The found values

lt(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods lt

mad(*axis=None*, *skipna=None*, *level=None*)
Return the mean absolute deviation of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mad : Series or DataFrame (if level specified)

mask(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

`DataFrame.where()`

max(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the maximum of the values in the object. If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

max : Series or DataFrame (if level specified)

mean(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mean : Series or DataFrame (if level specified)

median(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

median : Series or DataFrame (if level specified)

melt(*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *col_level=None*)

"Unpivots" a DataFrame from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

New in version 0.20.0.

frame : DataFrame id_vars : tuple, list, or ndarray, optional

Column(s) to use as identifier variables.

value_vars [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name [scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name [scalar, default 'value'] Name to use for the 'value' column.

col_level [int or string, optional] If columns are a MultiIndex then use this level to melt.

melt pivot_table DataFrame.pivot

```
>>> import pandas as pd
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> df.melt(id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

memory_usage(*index=True*, *deep=False*)

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

index [bool, default True] Specifies whether to include the memory usage of the DataFrame's index in returned Series. If `index=True` the memory usage of the index the first item in the output.

deep [bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

sizes [Series] A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

numpy.ndarray.nbytes [Total bytes consumed by the elements of an] ndarray.

Series.memory_usage : Bytes consumed by a Series. pandas.Categorical : Memory-efficient array for string values with

many repeated values.

DataFrame.info : Concise summary of a DataFrame.

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64  complex128 object  bool
0      1      1.0      (1+0j)      1  True
1      1      1.0      (1+0j)      1  True
2      1      1.0      (1+0j)      1  True
3      1      1.0      (1+0j)      1  True
4      1      1.0      (1+0j)      1  True
```

```
>>> df.memory_usage()
Index            80
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64         40000
float64       40000
complex128    80000
object        40000
```

(continues on next page)

(continued from previous page)

```
bool           5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index             80
int64          40000
float64        40000
complex128     80000
object        160000
bool            5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5168
```

merge(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *sort=False*, *suffixes=('_x', '_y')*, *copy=True*, *indicator=False*, *validate=None*)

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

right : DataFrame how : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys

on [label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [boolean, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

right_index [boolean, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index

sort [boolean, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword)

suffixes [2-length sequence (tuple, list, ...)] Suffix to apply to overlapping column names in the left and right side, respectively

copy [boolean, default True] If False, do not copy data unnecessarily

indicator [boolean or string, default False] If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in 'left' DataFrame, "right_only" for observations whose merge key only appears in 'right' DataFrame, and "both" if the observation's merge key is found in both.

validate [string, default None] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

New in version 0.21.0.

Support for specifying index levels as the *on*, *left_on*, and *right_on* parameters was added in version 0.23.0

```
>>> A              >>> B
    lkey value         rkey value
0   foo  1         0   foo  5
1   bar  2         1   bar  6
2   baz  3         2   qux  7
3   foo  4         3   bar  8
```

```
>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0  foo   1        foo   5
1  foo   4        foo   5
2  bar   2        bar   6
3  bar   2        bar   8
4  baz   3        NaN   NaN
5  NaN   NaN      qux   7
```

merged [DataFrame] The output type will the be same as 'left', if it is a subclass of DataFrame.

merge_ordered merge_asof DataFrame.join

merge_results(*others*)

Merges results of type *EpitopePredictionResult* and returns the merged result

Parameters **others** (list(*EpitopePredictionResult*)/*EpitopePredictionResult*) – Another (list of) :class:'~Fred2.Core.Result.EpitopePredictionResult'(s)

Returns A new merged *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

min(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the minimum of the values in the object. If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min : Series or DataFrame (if level specified)

mod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmod

mode(*axis=0*, *numeric_only=False*)

Gets the mode(s) of each element along the axis selected. Adds a row for each mode per label, fills in gaps with nan.

Note that there could be multiple values returned for the selected axis (when more than one item share the maximum frequency), which is the reason why a dataframe is returned. If you want to impute missing values with the mode in a dataframe `df`, you can just do this: `df.fillna(df.mode().iloc[0])`

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index' : get mode of each column
- 1 or 'columns' : get mode of each row

numeric_only [boolean, default False] if True, only apply to numeric columns

modes : DataFrame (sorted)

```
>>> df = pd.DataFrame({'A': [1, 2, 1, 2, 1, 2, 3]})
>>> df.mode()
   A
0  1
1  2
```

mul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

multiply(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

ndim

Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

ndarray.ndim

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

ne(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods ne

nlargest(*n*, *columns*, *keep='first'*)

Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant.

n [int] Number of rows to return.

columns [label or list of labels] Column label(s) to order by.

keep [{'first', 'last'}, default 'first'] Where there are duplicate values:

- *first* : prioritize the first occurrence(s)
- *last* : prioritize the last occurrence(s)

DataFrame The first *n* rows ordered by the given columns in descending order.

DataFrame.nsmallest [Return the first *n* rows ordered by *columns* in] ascending order.

DataFrame.sort_values : Sort DataFrame by the values DataFrame.head : Return the first *n* rows without re-ordering.

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 10, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df
    a  b    c
0   1  a  1.0
1  10  b  2.0
2   8  d  NaN
3  10  c  3.0
4  -1  e  4.0
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column "a".

```
>>> df.nlargest(3, 'a')
    a  b    c
1  10  b  2.0
3  10  c  3.0
2   8  d  NaN
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'a', keep='last')
    a  b    c
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

To order by the largest values in column "a" and then "c", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['a', 'c'])
    a  b    c
```

(continues on next page)

(continued from previous page)

```
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

Attempting to use `nlargest` on non-numeric dtypes will raise a `TypeError`:

```
>>> df.nlargest(3, 'b')
Traceback (most recent call last):
TypeError: Column 'b' has dtype object, cannot use method 'nlargest'
```

notna()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

notnull()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

nsmallest(*n*, *columns*, *keep='first'*)

Get the rows of a DataFrame sorted by the *n* smallest values of *columns*.

n [int] Number of items to retrieve

columns [list or str] Column name or names to order by

keep [{'first', 'last'}, default 'first'] Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

DataFrame

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 11, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nsmallest(3, 'a')
   a  b   c
4 -1  e   4
0  1  a   1
2  8  d NaN
```

nunique (*axis=0*, *dropna=True*)

Return Series with number of distinct observations over requested axis.

New in version 0.20.0.

axis : {0 or 'index', 1 or 'columns'}, default 0 dropna : boolean, default True

Don't include NaN in the counts.

nunique : Series

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 1, 1]})
>>> df.nunique()
A    3
B    1
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
```

pct_change (*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs** Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

chg [Series or DataFrame] The same type as the calling object.

Series.diff : Compute the difference of two elements in a Series. DataFrame.diff : Compute the difference of two elements in a DataFrame. Series.shift : Shift the index by some number of periods. DataFrame.shift : Shift the index by some number of periods.

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
```

(continues on next page)

(continued from previous page)

```
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                   FR        GR        IT
1980-01-01        NaN       NaN       NaN
1980-02-01   0.013810  0.013684  0.006549
1980-03-01   0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...      '2016': [1769950, 30586265],
...      '2015': [1500923, 40912316],
...      '2014': [1371819, 41403351]},
...      index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pipe (*func*, **args*, ***kwargs*)

Apply func(self, *args, **kwargs)

func [function] function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the NDFrame.

args [iterable, optional] positional arguments passed into `func`.

kwargs [mapping, optional] a dictionary of keyword arguments passed into `func`.

object : the return type of `func`.

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pandas.DataFrame.apply pandas.DataFrame.applymap pandas.Series.map

pivot (*index=None*, *columns=None*, *values=None*)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the User Guide for more on reshaping.

index [string or object, optional] Column to use to make new frame's index. If None, uses existing index.

columns [string or object] Column to use to make new frame's columns.

values [string, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Changed in version 0.23.0: Also accept list of column names.

DataFrame Returns reshaped DataFrame.

ValueError: When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

DataFrame.pivot_table [generalization of pivot that can handle] duplicate values for one index/column pair.

DataFrame.unstack [pivot based on the index values instead of a] column.

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
```

(continues on next page)

(continued from previous page)

```
...                    "baz": [1, 2, 3, 4]})
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

pivot_table(*values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*, *dropna=True*, *margins_name='All'*)

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame

values : column to aggregate, optional index : column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

columns [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

aggfunc [function, list of functions, dict, default numpy.mean] If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions

fill_value [scalar, default None] Value to replace missing values with

margins [boolean, default False] Add all row / columns (e.g. for subtotal / grand totals)

dropna [boolean, default True] Do not include columns whose entries are all NaN

margins_name [string, default 'All'] Name of the row / column that will contain the totals when margins is True.

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
>>> df
     A    B      C  D
0  foo  one  small  1
1  foo  one  large  2
2  foo  one  large  2
```

(continues on next page)

(continued from previous page)

```
3  foo  two  small  3
4  foo  two  small  3
5  bar  one  large  4
6  bar  one  small  5
7  bar  two  small  6
8  bar  two  large  7
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': [min, max, np.mean]})
>>> table
                D   E
             mean max median min
A   C
bar large  5.500000  16   14.5  13
    small  5.500000  15   14.5  14
foo large  2.000000  10    9.5   9
    small  2.333333  12   11.0   8
```

table : DataFrame

DataFrame.pivot [pivot without aggregation that can handle] non-numeric data

plot

alias of `pandas.plotting._core.FramePlotMethods`

pop(*item*)

Return item and drop from frame. Raise KeyError if not found.

item [str] Column label to be popped

popped : Series

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
```

(continues on next page)

(continued from previous page)

```
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

pow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rpow

prod(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

product (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

quantile(*q=0.5*, *axis=0*, *numeric_only=True*, *interpolation='linear'*)

Return values at the given quantile over requested axis, a la numpy.percentile.

q [float or array-like, default 0.5 (50% quantile)] 0 <= q <= 1, the quantile(s) to compute

axis [{0, 1, 'index', 'columns'} (default 0)] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

numeric_only [boolean, default True] If False, the quantile of datetime and timedelta data will be computed as well

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.18.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

quantiles : Series or DataFrame

- If `q` is an array, a DataFrame will be returned where the index is `q`, the columns are the columns of self, and the values are the quantiles.
- If `q` is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                      columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
                       'B': [pd.Timestamp('2010'),
                             pd.Timestamp('2011')],
                       'C': [pd.Timedelta('1 days'),
                             pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
```

(continues on next page)

(continued from previous page)

```
C        1 days 12:00:00
Name: 0.5, dtype: object
```

pandas.core.window.Rolling.quantile

query(*expr*, *inplace=False*, ***kwargs*)

Query the columns of a frame with a boolean expression.

expr [string] The query string to evaluate. You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

inplace [bool] Whether the query should modify the data in place or return a modified copy

New in version 0.18.0.

kwargs [dict] See the documentation for `pandas.eval()` for complete details on the keyword arguments accepted by `DataFrame.query()`.

q : DataFrame

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level `pandas.eval()` function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the `query` documentation in indexing.

pandas.eval DataFrame.eval

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = pd.DataFrame(randn(10, 2), columns=list('ab'))
>>> df.query('a > b')
>>> df[df.a > df.b]  # same result as the previous expression
```

radd(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.add

rank (*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)
Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values

axis [{0 or 'index', 1 or 'columns'}, default 0] index to direct ranking

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

ranks : same type as caller

rdiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

reindex(***kwargs*)
Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like, optional] New labels / index to conform the axis specified by 'axis' to.

index, columns [array-like, optional (should be specified using keywords)] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis [int or str, optional] Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...       'http_status': [200,200,404,404,301],
...       'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...        index=index)
>>> df
            http_status  response_time
Firefox             200           0.04
Chrome              200           0.02
Safari              404           0.07
IE10                404           0.08
Konqueror           301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
```

(continues on next page)

(continued from previous page)

```
IE10                  404          0.08
Chrome                200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to backpropagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the `method` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29     100
2009-12-30     100
2009-12-31     100
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

Please note that the `NaN` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the `NaN` values present in the original dataframe, use the `fillna()` method.

See the user guide for more.

reindexed : DataFrame

`reindex_axis` (*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=nan*)
Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis : {0 or 'index', 1 or 'columns'} method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional

Method to use for filling holes in reindexed DataFrame:

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

reindex, reindex_like

reindexed : DataFrame

reindex_like(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)
Return an object with matching indices to myself.

other : Object method : string or None copy : boolean, default True limit : int, default None

Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

reindexed : same as input

rename(***kwargs*)
Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the user guide for more.

mapper, index, columns [dict-like or function, optional] dict-like or functions transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

axis [int or str, optional] Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

copy [boolean, default True] Also copy underlying data

inplace [boolean, default False] Whether to return a new DataFrame. If True then value of copy is ignored.

level [int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

renamed : DataFrame

pandas.DataFrame.rename_axis

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
```

(continues on next page)

(continued from previous page)

```
1  2  5
2  3  6
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

rename_axis(*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter the name of the index or columns.

mapper [scalar, list-like, optional] Value to set as the axis name attribute.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis.

copy [boolean, default True] Also copy underlying data.

inplace [boolean, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

renamed [Series, DataFrame, or None] The same type as the caller or None if *inplace* is True.

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

pandas.Series.rename : Alter Series index labels or name pandas.DataFrame.rename : Alter DataFrame index labels or name pandas.Index.rename : Set new names on index

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s.rename_axis("foo")
foo
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")
     A  B
foo
0    1  4
1    2  5
2    3  6
```

```
>>> df.rename_axis("bar", axis="columns")
bar  A  B
0    1  4
1    2  5
2    3  6
```

reorder_levels (*order*, *axis=0*)

Rearrange index levels using input order. May not drop or duplicate levels

order [list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).

axis [int] Where to reorder levels.

type of caller (new object)

replace (*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)

Replace values given in *to_replace* with *value*.

Values of the DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

to_replace [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be `None` to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:

- This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

value [scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace [boolean, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

regex [bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method [{'pad', 'ffill', 'bfill', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

DataFrame.fillna : Fill NA values DataFrame.where : Replace values based on boolean condition Series.str.replace : Simple string replacement.

DataFrame Object after replacement.

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Scalar ‘to_replace‘ and ‘value‘

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like ‘to_replace‘

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
2    3
3    3
4    4
dtype: int64
```

dict-like ‘to_replace‘

```
>>> df.replace({0: 10, 1: 100})
     A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
     A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
     A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression 'to_replace'

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
      A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$':'new', 'foo':'xyz'})
      A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
      A    B
0   new  abc
1   new  new
2  bait  xyz
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
```

(continues on next page)

(continued from previous page)

```
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the pecularities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value=None` and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

resample(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

rule [string] the offset string or object representing target conversion

axis : int, optional, default 0 closed : {'right', 'left'}

Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label [{'right', 'left'}] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention [{'start', 'end', 's', 'e'}] For PeriodIndex only, controls whether to use the start or end of *rule*

kind: {'timestamp', 'period'}, optional Pass 'timestamp' to convert the resulting index to a `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`. By default the input representation is retained.

loffset [timedelta] Adjust the resampled time labels

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

on [string, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [string or int, optional] For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

Resampler object

See the user guide for more.

To learn more about the offset strings, please see this link.

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5] #select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like)+5
```

```
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))
>>> s
2012    1
```

(continues on next page)

(continued from previous page)

```
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using 'start' *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
                     a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                       columns=['a', 'b', 'c', 'd'],
                       index=pd.MultiIndex.from_product([time, [1, 2]])
                       )
>>> df2.resample('3T', level=0).sum()
                     a  b   c   d
2000-01-01 00:00:00  0  6  12  18
2000-01-01 00:03:00  0  4   8  12
```

groupby : Group by mapping, function, label, or list of labels.

reset_index(*level=None*, *drop=False*, *inplace=False*, *col_level=0*, *col_fill=''*)

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level_0' (if 'index' is already taken) will be used.

level [int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default

drop [boolean, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

col_level [int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill [object, default ''] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

resetted : DataFrame

```
>>> df = pd.DataFrame([('bird',    389.0),
...                    ('bird',     24.0),
...                    ('mammal',   80.5),
...                    ('mammal', np.nan)],
...                   index=['falcon', 'parrot', 'lion', 'monkey'],
...                   columns=('class', 'max_speed'))
>>> df
         class  max_speed
falcon    bird      389.0
parrot    bird       24.0
lion    mammal       80.5
monkey  mammal        NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
    class  max_speed
0    bird      389.0
1    bird       24.0
2  mammal       80.5
3  mammal        NaN
```

You can also use *reset_index* with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                    ('bird', 'parrot'),
...                                    ('mammal', 'lion'),
...                                    ('mammal', 'monkey')],
```

(continues on next page)

(continued from previous page)

```
...                                       names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                      ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
...                    ( 24.0, 'fly'),
...                    ( 80.5, 'run'),
...                    (np.nan, 'jump')],
...                   index=index,
...                   columns=columns)
>>> df
               speed species
                 max    type
class  name
bird   falcon  389.0     fly
       parrot   24.0     fly
mammal lion     80.5     run
       monkey    NaN    jump
```

If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
         class  speed species
                  max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
                speed species
         class    max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

When the index is inserted under another level, we can specify under which one with the parameter *col_fill*:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
              species  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

If we specify a nonexistent level for *col_fill*, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
                genus  speed species
                class    max    type
```

(continues on next page)

(continued from previous page)

```
name
falcon          bird  389.0     fly
parrot          bird   24.0     fly
lion          mammal   80.5     run
monkey        mammal    NaN    jump
```

rfloordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.floordiv

rmod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mod

rmul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mul

rolling(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculations.

New in version 0.18.0.

window [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. This is new in 0.19.0

min_periods [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

center [boolean, default False] Set the labels at the center of the window.

win_type [string, default None] Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

on [string, optional] For a DataFrame, column on which to calculate the rolling window, rather than the index

closed [string, default None] Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

axis : int or string, default 0

a Window or Rolling sub-classed for the particular operation

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
     B
0  NaN
1  1.0
```

(continues on next page)

(continued from previous page)

```
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                   index = [pd.Timestamp('20130101 09:00:00'),
...                            pd.Timestamp('20130101 09:00:02'),
...                            pd.Timestamp('20130101 09:00:03'),
...                            pd.Timestamp('20130101 09:00:05'),
...                            pd.Timestamp('20130101 09:00:06')])
```

```
>>> df
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```
>>> df.rolling('2s').sum()
                       B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

To learn more about the offsets & frequency strings, please see this link.

The recognized win_types are:

- boxcar
- triang
- blackman
- hamming
- bartlett
- parzen
- bohman
- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general_gaussian (needs power, width)
- slepian (needs width).

If win_type=None all points are evenly weighted. To learn more about different window types see scipy.signal window functions.

expanding : Provides expanding transformations. ewm : Provides exponential weighted functions

round(*decimals=0*, **args*, ***kwargs*)

Round a DataFrame to a variable number of decimal places.

decimals [int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

```
>>> df = pd.DataFrame(np.random.random([3, 3]),
...     columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
>>> df
                A         B         C
first    0.028208  0.992815  0.173891
second   0.038683  0.645646  0.577595
third    0.877076  0.149370  0.491027
>>> df.round(2)
           A     B     C
first   0.03  0.99  0.17
second  0.04  0.65  0.58
third   0.88  0.15  0.49
>>> df.round({'A': 1, 'C': 2})
          A         B     C
first   0.0  0.992815  0.17
second  0.0  0.645646  0.58
third   0.9  0.149370  0.49
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
          A  B     C
first   0.0  1  0.17
```

(continues on next page)

(continued from previous page)

```
second  0.0  1  0.58
third   0.9  0  0.49
```

DataFrame object

numpy.around Series.round

rpow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.pow

rsub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
```

(continues on next page)

(continued from previous page)

```
...                   index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.sub

rtruediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*)

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

n [int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac [float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace [boolean, optional] Sample with or without replacement. Default = False.

weights [str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. inf and -inf values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

A new object of same type as caller.

Generate an example `Series` and `DataFrame`:

```
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0   -0.038497
1    1.820773
2   -0.972766
3   -1.598270
4   -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
          A         B         C         D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661
```

Next extract a random sample from both of these objects...

3 random elements from the `Series`:

```
>>> s.sample(n=3)
27   -0.994689
55   -1.049016
67   -0.224565
dtype: float64
```

And a random 10% of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.1, replace=True)
           A         B         C         D
35  1.981780  0.142106  1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640  0.217116
40  0.823173 -0.078816  1.009536  1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
6  -0.148339  0.832938  1.787600 -1.383767
```

You can use *random state* for reproducibility:

```
>>> df.sample(random_state=1)
A         B         C         D
37 -2.027662  0.103611  0.237496 -0.165867
43 -0.259323 -0.583426  1.516140 -0.479118
12 -1.686325 -0.579510  0.985195 -0.460286
8   1.167946  0.429082  1.215742 -1.636041
9   1.197475 -0.864188  1.554031 -1.505264
```

select (*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

crit [function] To be called on each index (label). Should return True or False

axis : int

selection : type of caller

select_dtypes(*include=None*, *exclude=None*)

Return a subset of the DataFrame's columns based on the column dtypes.

include, exclude [scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

ValueError

- If both of `include` and `exclude` are empty
- If `include` and `exclude` have overlapping elements
- If any kind of string dtype is passed in.

subset [DataFrame] The subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.

- To select all *numeric* types, use `np.number` or `'number'`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the numpy dtype hierarchy
- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`
- To select timedeltas, use `np.timedelta64`, `'timedelta'` or `'timedelta64'`
- To select Pandas categorical dtypes, use `'category'`
- To select Pandas datetimetz dtypes, use `'datetimetz'` (new in 0.20.0) or `'datetime64[ns, tz]'`

```
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
        a      b  c
0       1   True  1.0
1       2  False  2.0
2       1   True  1.0
3       2  False  2.0
4       1   True  1.0
5       2  False  2.0
```

```
>>> df.select_dtypes(include='bool')
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

```
>>> df.select_dtypes(include=['float64'])
   c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int'])
       b    c
0   True  1.0
1  False  2.0
2   True  1.0
3  False  2.0
4   True  1.0
5  False  2.0
```

sem(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

sem : Series or DataFrame (if level specified)

set_axis(*labels*, *axis=0*, *inplace=None*)
Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

labels [list-like, Index] The values for the new index.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

pandas.DataFrame.rename_axis : Alter the name of the index or columns.

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

set_index (*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

keys : column label or list of column labels / arrays drop : boolean, default True

Delete columns to be used as the new index

append [boolean, default False] Whether to append columns to existing index

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

verify_integrity [boolean, default False] Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale':[55, 40, 84, 31]})
   month  sale  year
0  1      55    2012
1  4      40    2014
2  7      84    2013
3  10     31    2014
```

Set the index to become the 'month' column:

```
>>> df.set_index('month')
       sale  year
month
1      55    2012
4      40    2014
7      84    2013
10     31    2014
```

Create a multi-index using columns 'year' and 'month':

```
>>> df.set_index(['year', 'month'])
            sale
year  month
2012  1     55
2014  4     40
2013  7     84
2014  10    31
```

Create a multi-index using a set of values and a column:

```
>>> df.set_index([[1, 2, 3, 4], 'year'])
         month  sale
   year
1  2012  1      55
2  2014  4      40
3  2013  7      84
4  2014  10     31
```

dataframe : DataFrame

set_value(*index*, *col*, *value*, *takeable=False*)

Put single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label value : scalar value takeable : interpret the index/col as indexers, default False

frame [DataFrame] If label pair is contained, will be reference to calling DataFrame, otherwise a new object

shape
Return a tuple representing the dimensionality of the DataFrame.

ndarray.shape

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

shift (*periods=1*, *freq=None*, *axis=0*)
Shift index by desired number of periods with an optional time freq

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, optional] Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

axis : {0 or 'index', 1 or 'columns'}

If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data.

shifted : DataFrame

size
Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

ndarray.size

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

skew (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

skew : Series or DataFrame (if level specified)

slice_shift (*periods=1*, *axis=0*)
Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

periods [int] Number of periods to move, can be positive or negative

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

shifted : same type as caller

sort_index (*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)
Sort object by labels (along an axis)

axis : index, columns to direct sorting level : int or level name or list of ints or list of level names

if not None, sort on values in specified index level(s)

ascending [boolean, default True] Sort ascending vs. descending

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining [bool, default True] if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

sorted_obj : DataFrame

sort_values (*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)
Sort by the values along either axis

by [str or list of str] Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels

Changed in version 0.23.0: Allow specifying index or column level names.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis to be sorted

ascending [bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end

sorted_obj : DataFrame

```
>>> df = pd.DataFrame({
...     'col1' : ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2' : [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
```

(continues on next page)

(continued from previous page)

```
... })
>>> df
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
3   NaN  8    4
4   D    7    2
5   C    4    3
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
    col1 col2 col3
1   A    1    1
0   A    2    0
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
    col1 col2 col3
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
3   NaN  8    4
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
    col1 col2 col3
3   NaN  8    4
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
```

sortlevel (*level=0*, *axis=0*, *ascending=True*, *inplace=False*, *sort_remaining=True*)

Sort multilevel index by chosen axis and primary level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order).

Deprecated since version 0.20.0: Use `DataFrame.sort_index()`

level : int axis : {0 or 'index', 1 or 'columns'}, default 0 ascending : boolean, default True inplace : boolean, default False

Sort the DataFrame without creating a new instance

sort_remaining [boolean, default True] Sort by the other levels too.

sorted : DataFrame

DataFrame.sort_index(level=...)

squeeze (*axis=None*)

Squeeze length 1 dimensions.

axis [None, integer or string axis name, optional] The axis to squeeze if 1-sized.

New in version 0.20.0.

scalar if 1-sized, else original object

stack (*level=-1*, *dropna=True*)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

The new index levels are sorted.

level [int, str, list, default -1] Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

dropna [bool, default True] Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of index and column values that are missing from the original dataframe. See Examples section.

DataFrame or Series Stacked dataframe or series.

DataFrame.unstack [Unstack prescribed level(s) from index axis] onto column axis.

DataFrame.pivot [Reshape dataframe from long format to wide] format.

DataFrame.pivot_table [Create a spreadsheet-style pivot table] as a DataFrame.

The function is named by analogy with a collection of books being re-organised from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of of each other (in the index of the dataframe).

Single level columns

```
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
     weight height
cat       0      1
dog       2      3
>>> df_single_level_cols.stack()
cat  weight    0
     height    1
dog  weight    2
     height    3
dtype: int64
```

Multi level columns: simple case

```
>>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol1)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
     weight
         kg    pounds
cat       1         2
dog       2         4
>>> df_multi_level_cols1.stack()
            weight
cat kg           1
    pounds       2
dog kg           2
    pounds       4
```

Missing values

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
    weight height
        kg      m
cat    1.0    2.0
dog    3.0    4.0
>>> df_multi_level_cols2.stack()
        height  weight
cat kg     NaN     1.0
    m      2.0     NaN
dog kg     NaN     3.0
    m      4.0     NaN
```

Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0)
             kg    m
cat height  NaN  2.0
    weight  1.0  NaN
dog height  NaN  4.0
    weight  3.0  NaN
>>> df_multi_level_cols2.stack([0, 1])
cat  height  m     2.0
     weight  kg    1.0
dog  height  m     4.0
     weight  kg    3.0
dtype: float64
```

Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
    weight height
        kg      m
cat    NaN    1.0
dog    2.0    3.0
>>> df_multi_level_cols3.stack(dropna=False)
        height  weight
cat kg     NaN     NaN
    m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
>>> df_multi_level_cols3.stack(dropna=True)
        height  weight
cat m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
```

std(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

std : Series or DataFrame (if level specified)

style
Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

pandas.io.formats.style.Styler

sub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

subtract(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the sum of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

sum : Series or DataFrame (if level specified)

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

swapaxes (*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately

y : same as input

swaplevel (*i=-2*, *j=-1*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

swapped : type of caller (new object)

Changed in version 0.18.1: The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

tail (*n=5*)

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

n [int, default 5] Number of rows to select.

type of caller The last *n* rows of the caller object.

pandas.DataFrame.head : The first *n* rows of the caller object.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

take(*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

indices [array-like] An array of ints indicating which positions to take.

axis [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

taken [type of caller] An array-like containing the elements taken from the object.

DataFrame.loc : Select a subset of a DataFrame by labels. DataFrame.iloc : Select a subset of a DataFrame by positions. numpy.take : Take elements from an array along an axis.

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                    columns=['name', 'class', 'max_speed'],
...                    index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

to_clipboard(*excel=True*, *sep=None*, ***kwargs*)

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

excel [bool, default True]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

sep [str, default `'\t'`] Field delimiter.

****kwargs** These parameters will be passed to DataFrame.to_csv.

DataFrame.to_csv [Write a DataFrame to a comma-separated values] (csv) file.

read_clipboard : Read text from clipboard and pass to read_table.

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

to_csv(*path_or_buf=None*, *sep=', '*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *encoding=None*, *compression=None*, *quoting=None*, *quotechar='"'*, *line_terminator='\n'*, *chunksize=None*, *tupleize_cols=None*, *date_format=None*, *doublequote=True*, *escapechar=None*, *decimal='.'*)

Write DataFrame to a comma-separated values (csv) file

path_or_buf [string or file handle, default None] File path or object, if None is provided the result is returned as a string.

sep [character, default ‘,’] Field delimiter for the output file.

na_rep [string, default ‘’] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R

mode [str] Python write mode, default ‘w’

encoding [string, optional] A string representing the encoding to use in the output file, defaults to ‘ascii’ on Python 2 and ‘utf-8’ on Python 3.

compression [string, optional] A string representing the compression to use in the output file. Allowed values are ‘gzip’, ‘bz2’, ‘zip’, ‘xz’. This input is only used when the first argument is a filename.

line_terminator [string, default `'\n'`] The newline character or character sequence to use in the output file

quoting [optional constant from csv module] defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric

quotechar [string (length 1), default ‘"’] character used to quote fields

doublequote [boolean, default True] Control quoting of *quotechar* inside a field

escapechar [string (length 1), default None] character used to escape *sep* and *quotechar* when appropriate

chunksize [int or None] rows to write at a time

tupleize_cols [boolean, default False] Deprecated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

Write MultiIndex columns as a list of tuples (if True) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if False).

date_format [string, default None] Format string for datetime objects

decimal: string, default '.' Character recognized as decimal separator. E.g. use ',' for European data

to_dense()
Return dense representation of NDFrame (as opposed to sparse)

to_dict(*orient='dict'*, *into=<type 'dict'>*)
Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

orient [str {'dict', 'list', 'series', 'split', 'records', 'index'}] Determines the type of the values of the dictionary.

- 'dict' (default) : dict like {column -> {index -> value}}
- 'list' : dict like {column -> [values]}
- 'series' : dict like {column -> Series(values)}
- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'records' : list like [{column -> value}, ... , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

into [class, default dict] The collections.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

New in version 0.21.0.

result : collections.Mapping like {column -> {index -> value}}

DataFrame.from_dict: create a DataFrame from a dictionary DataFrame.to_json: convert a DataFrame to JSON format

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   col1  col2
a     1   0.50
b     2   0.75
>>> df.to_dict()
{'col1': {'a': 1, 'b': 2}, 'col2': {'a': 0.5, 'b': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': a    1
         b    2
         Name: col1, dtype: int64,
 'col2': a    0.50
         b    0.75
         Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['a', 'b'], 'columns': ['col1', 'col2'],
 'data': [[1.0, 0.5], [2.0, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1.0, 'col2': 0.5}, {'col1': 2.0, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'a': {'col1': 1.0, 'col2': 0.5}, 'b': {'col1': 2.0, 'col2': 0.75}}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('a', 1), ('b', 2)])),
             ('col2', OrderedDict([('a', 0.5), ('b', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1.0, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2.0, 'col2': 0.75})]
```

to_excel(*excel_writer*, *sheet_name='Sheet1'*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *startrow=0*, *startcol=0*, *engine=None*, *merge_cells=True*, *encoding=None*, *inf_rep='inf'*, *verbose=True*, *freeze_panes=None*)

Write DataFrame to an excel sheet

excel_writer [string or ExcelWriter object] File path or existing ExcelWriter

sheet_name [string, default 'Sheet1'] Name of sheet which will contain DataFrame

na_rep [string, default ''] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow : upper left cell row to dump data frame

startcol : upper left cell column to dump data frame

engine [string, default None] write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

merge_cells [boolean, default True] Write MultiIndex and Hierarchical Rows as merged cells.

encoding: string, default None encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep [string, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel)

freeze_panes [tuple of integer (length 2), default None] Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

For compatibility with to_csv, to_excel serializes lists and dicts to strings before writing.

to_feather(*fname*)
write out the binary feather-format for DataFrames

New in version 0.20.0.

fname [str] string file path

to_gbq(*destination_table*, *project_id*, *chunksize=None*, *verbose=None*, *reauth=False*, *if_exists='fail'*, *private_key=None*, *auth_local_webserver=False*, *table_schema=None*)
Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

Authentication to the Google BigQuery service is via OAuth 2.0.

- If `private_key` is provided, the library loads the JSON service account credentials and uses those to authenticate.
- If no `private_key` is provided, the library tries application default credentials.
- If application default credentials are not found or cannot be used with BigQuery, the library authenticates with user account credentials. In this case, you will be asked to grant permissions for product name 'pandas GBQ'.

destination_table [str] Name of table to be written, in the form 'dataset.tablename'.

project_id [str] Google BigQuery Account project ID.

chunksize [int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

reauth [bool, default False] Force Google BigQuery to reauthenticate the user. This is useful if multiple accounts are used.

if_exists [str, default 'fail'] Behavior when the destination table exists. Value can be one of:

'fail' If table exists, do nothing.

'replace' If table exists, drop it, recreate it, and insert data.

'append' If table exists, insert data. Create if does not exist.

private_key [str, optional] Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

auth_local_webserver [bool, default False] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

table_schema [list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'},...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

New in version 0.3.1 of pandas-gbq.

verbose [boolean, deprecated] *Deprecated in Pandas-GBQ 0.4.0.* Use the logging module to adjust verbosity instead.

pandas_gbq.to_gbq : This function in the pandas-gbq library. pandas.read_gbq : Read a DataFrame from Google BigQuery.

to_hdf (*path_or_buf*, *key*, ***kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the user guide.

path_or_buf [str or pandas.HDFStore] File path or HDFStore object.

key [str] Identifier for the group in the store.

mode [{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

format [{'fixed', 'table'}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

append [bool, default False] For Table formats, append the input data to the existing.

data_columns [list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See io.hdf5-query-data-columns. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

errors [str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

DataFrame.read_hdf : Read from HDF file. DataFrame.to_parquet : Write a DataFrame to the binary parquet format. DataFrame.to_sql : Write to a sql table. DataFrame.to_feather : Write out feather-format for DataFrames. DataFrame.to_csv : Write out to a csv file.

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

to_html (*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *bold_rows=True*, *classes=None*, *escape=True*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*, *notebook=False*, *decimal='.'*, *border=None*, *table_id=None*)

Render a DataFrame as an HTML table.

to_html-specific options:

bold_rows [boolean, default True] Make the row labels bold in the output

classes [str or list or tuple, default None] CSS class(es) to apply to the resulting html table

escape [boolean, default True] Convert the characters <, >, and & to HTML-safe sequences.

max_rows [int, optional] Maximum number of rows to show before truncating. If None, show all.

max_cols [int, optional] Maximum number of columns to show before truncating. If None, show all.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe

New in version 0.18.0.

border [int] A `border=border` attribute is included in the opening <*table*> tag. Default `pd.options.html.border`.

New in version 0.19.0.

table_id [str, optional] A css id is included in the opening <*table*> tag if specified.

New in version 0.23.0.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] whether to print column labels, default True

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_json(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression=None*, *index=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

path_or_buf [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'

  - allowed values are: {‘split’,’records’,’index’}
- DataFrame
  - default is ‘columns’
  - allowed values are: {‘split’,’records’,’index’,’columns’,’values’}
- The format of the JSON string
  - ‘split’ : dict like {‘index’ -> [index], ‘columns’ -> [columns], ‘data’ -> [values]}
  - ‘records’ : list like [{column -> value}, … , {column -> value}]
  - ‘index’ : dict like {index -> {column -> value}}
  - ‘columns’ : dict like {column -> {index -> value}}
  - ‘values’ : just the values array
  - ‘table’ : dict like {‘schema’: {schema}, ‘data’: {data}} describing the data, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

date_format [{None, ‘epoch’, ‘iso’}] Type of date conversion. ‘epoch’ = epoch milliseconds, ‘iso’ = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is ‘iso’. For all other orients, the default is ‘epoch’.

double_precision [int, default 10] The number of decimal places to use when encoding floating point values.

force_ascii [boolean, default True] Force encoded string to be ASCII.

date_unit [string, default ‘ms’ (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’, ‘ns’ for second, millisecond, microsecond, and nanosecond respectively.

default_handler [callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines [boolean, default False] If ‘orient’ is ‘records’ write out line delimited json format. Will throw ValueError if incorrect ‘orient’ since others are not list like.

New in version 0.19.0.

compression [{None, ‘gzip’, ‘bz2’, ‘zip’, ‘xz’}] A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

index [boolean, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is ‘split’ or ‘table’.

New in version 0.23.0.

pandas.read_json

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]}'
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

to_latex(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=False*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires \usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

to_latex-specific options:

bold_rows [boolean, default False] Make the row labels bold in the output

column_format [str, default None] The columns format as specified in LaTeX table format e.g 'rcl' for 3 columns

longtable [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a \usepackage{longtable} to your LaTeX preamble.

escape [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

encoding [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

multicolumn [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module.

New in version 0.20.0.

multirow [boolean, default False] Use multirow to enhance MultiIndex rows. Requires adding a \usepackage{multirow} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

New in version 0.20.0.

to_msgpack (*path_or_buf=None*, *encoding='utf-8'*, ***kwargs*)
msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

path [string File path, buffer-like, or None] if None, return generated string

append [boolean whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

to_panel ()
Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Deprecated since version 0.20.0.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

panel : Panel

to_parquet (*fname*, *engine='auto'*, *compression='snappy'*, ***kwargs*)
Write a DataFrame to the binary parquet format.

New in version 0.21.0.

This function writes the dataframe as a parquet file. You can choose different parquet backends, and have the option of compression. See the user guide for more details.

fname [str] String file path.

engine [{'auto', 'pyarrow', 'fastparquet'}, default 'auto'] Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

compression [{'snappy', 'gzip', 'brotli', None}, default 'snappy'] Name of the compression to use. Use `None` for no compression.

****kwargs** Additional arguments passed to the parquet library. See pandas io for more details.

read_parquet : Read a parquet file. DataFrame.to_csv : Write a csv file. DataFrame.to_sql : Write to a sql table. DataFrame.to_hdf : Write to hdf.

This function requires either the fastparquet or pyarrow library.

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gzip', compression='gzip')
>>> pd.read_parquet('df.parquet.gzip')
   col1  col2
```

(continues on next page)

(continued from previous page)

```
0      1      3
1      2      4
```

to_period (*freq=None*, *axis=0*, *copy=True*)

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

freq : string, default axis : {0 or 'index', 1 or 'columns'}, default 0

The axis to convert (the index by default)

copy [boolean, default True] If False then underlying input data is not copied

ts : TimeSeries with PeriodIndex

to_pickle (*path*, *compression='infer'*, *protocol=2*)

Pickle (serialize) object to file.

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

protocol [int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1]_ paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

read_pickle : Load pickled pandas object (or any object) from file. DataFrame.to_hdf : Write DataFrame to an HDF5 file. DataFrame.to_sql : Write DataFrame to a SQL database. DataFrame.to_parquet : Write a DataFrame to the binary parquet format.

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

to_records(*index=True*, *convert_datetime64=None*)

Convert DataFrame to a NumPy record array.

Index will be put in the 'index' field of the record array if requested.

index [boolean, default True] Include index in resulting record array, stored in 'index' field.

convert_datetime64 [boolean, default None] Deprecated since version 0.23.0.

Whether to convert the index to datetime.datetime if it is a DatetimeIndex.

y : numpy.recarray

DataFrame.from_records: convert structured or record ndarray to DataFrame.

numpy.recarray: ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   A     B
a  1  0.50
b  2  0.75
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

By default, timestamps are converted to *datetime.datetime*:

```
>>> df.index = pd.date_range('2018-01-01 09:00', periods=2, freq='min')
>>> df
                     A     B
2018-01-01 09:00:00  1  0.50
2018-01-01 09:01:00  2  0.75
>>> df.to_records()
rec.array([(datetime.datetime(2018, 1, 1, 9, 0), 1, 0.5 ),
           (datetime.datetime(2018, 1, 1, 9, 1), 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The timestamp conversion can be disabled so NumPy's datetime64 data type is used instead:

```
>>> df.to_records(convert_datetime64=False)
rec.array([('2018-01-01T09:00:00.000000000', 1, 0.5 ),
           ('2018-01-01T09:01:00.000000000', 2, 0.75)],
          dtype=[('index', '<M8[ns]'), ('A', '<i8'), ('B', '<f8')])
```

to_sparse(*fill_value=None*, *kind='block'*)

Convert to SparseDataFrame

fill_value : float, default NaN kind : {'block', 'integer'}

y : SparseDataFrame

to_sql(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1]_ are supported. Tables can be newly created, appended to, or overwritten.

name [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists [{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index [boolean, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize [int, optional] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

ValueError When the table already exists and *if_exists* is 'fail' (the default).

pandas.read_sql : read a DataFrame from a table

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
```

(continues on next page)

(continued from previous page)

```
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

to_stata(*fname*, *convert_dates=None*, *write_index=True*, *encoding='latin-1'*, *byteorder=None*, *time_stamp=None*, *data_label=None*, *variable_labels=None*, *version=114*, *convert_strl=None*)

Export Stata binary dta files.

fname [path (string), buffer or path object] string, path object (pathlib.Path or py._path.local.LocalPath) or object implementing a binary write() functions. If using a buffer then the buffer will not be automatically closed after the file data has been written.

convert_dates [dict] Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to 'tc'. Raises NotImplementedError if a datetime column has timezone information.

write_index [bool] Write the index to Stata dataset.

encoding [str] Default is latin-1. Unicode is not supported.

byteorder [str] Can be ">", "<", "little", or "big". default is *sys.byteorder*.

time_stamp [datetime] A datetime to use as file creation date. Default is the current time.

data_label [str] A label for the data set. Must be 80 characters or smaller.

variable_labels [dict] Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

New in version 0.19.0.

version [{114, 117}] Version to use in the output dta file. Version 114 can be used read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 114 limits string variables to 244 characters

or fewer while 117 allows strings with lengths up to 2,000,000 characters.

New in version 0.23.0.

convert_strl [list, optional] List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

New in version 0.23.0.

NotImplementedError

- If datetimes contain timezone information
- Column dtype is not representable in Stata

ValueError

- Columns listed in convert_dates are neither datetime64[ns] or datetime.datetime
- Column listed in convert_dates is not in DataFrame
- Categorical label contains more than 32,000 characters

New in version 0.19.0.

pandas.read_stata : Import Stata data files pandas.io.stata.StataWriter : low-level writer for Stata data files pandas.io.stata.StataWriter117 : low-level writer for version 117 files

```
>>> data.to_stata('./data_file.dta')
```

Or with dates

```
>>> data.to_stata('./date_data_file.dta', {2 : 'tw'})
```

Alternatively you can create an instance of the StataWriter class

```
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

With dates:

```
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
```

to_string(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *line_width=None*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*)

Render a DataFrame to a console-friendly tabular output.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_timestamp (*freq=None*, *how='start'*, *axis=0*, *copy=True*)
Cast to DatetimeIndex of timestamps, at *beginning* of period

freq [string, default frequency of PeriodIndex] Desired frequency

how [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default)

copy [boolean, default True] If false then underlying input data is not copied

df : DataFrame with DatetimeIndex

to_xarray()
Return an xarray object from the pandas object.

a DataArray for a Series a Dataset for a DataFrame a DataArray for higher dims

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
```

(continues on next page)

(continued from previous page)

```
   A    B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index    (index) int64 0 1 2
Data variables:
    A        (index) int64 1 1 2
    B        (index) object 'foo' 'bar' 'foo'
    C        (index) float64 4.0 5.0 6.0
```

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)}
                      ).set_index(['B','A'])
>>> df
         C
B   A
foo 1  4.0
bar 1  5.0
foo 2  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B        (B) object 'bar' 'foo'
  * A        (A) int64 1 2
Data variables:
    C        (B, A) float64 5.0 nan 4.0 6.0
```

```
>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                 items=list('ABCD'),
                 major_axis=pd.date_range('20130101', periods=3),
                 minor_axis=['first', 'second'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second
```

```
>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],
       [[ 6,  7],
        [ 8,  9],
        [10, 11]],
       [[12, 13],
```

(continues on next page)

(continued from previous page)

```
        [14, 15],
        [16, 17]],
       [[18, 19],
        [20, 21],
        [22, 23]]])
Coordinates:
  * items       (items) object 'A' 'B' 'C' 'D'
  * major_axis  (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03
↪# noqa
  * minor_axis  (minor_axis) object 'first' 'second'
```

See the xarray docs

transform(*func*, **args*, ***kwargs*)

Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values

New in version 0.20.0.

func [callable, string, dictionary, or list of string/callables] To apply to column

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

transformed : NDFrame

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
...                   index=pd.date_range('1/1/2000', periods=10))
df.iloc[3:7] = np.nan
```

```
>>> df.transform(lambda x: (x - x.mean()) / x.std())
                   A         B         C
2000-01-01  0.579457  1.236184  0.123424
2000-01-02  0.370357 -0.605875 -1.231325
2000-01-03  1.455756 -0.277446  0.288967
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -0.498658  1.274522  1.642524
2000-01-09 -0.540524 -1.012676 -0.828968
2000-01-10 -1.366388 -0.614710  0.005378
```

pandas.NDFrame.aggregate pandas.NDFrame.apply

transpose(**args*, ***kwargs*)

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
                0     1
name        Alice   Bob
score         9.5     8
employed    False  True
kids            0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

truediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

before [date, string, int] Truncate all rows before this index value.

after [date, string, int] Truncate all rows after this index value.

axis [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

copy [boolean, default is True,] Return a copy of the truncated section.

type of caller The truncated Series or DataFrame.

DataFrame.loc : Select a subset of a DataFrame by label. DataFrame.iloc : Select a subset of a DataFrame by position.

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
```

(continues on next page)

(continued from previous page)

```
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

tshift (*periods=1*, *freq=None*, *axis=0*)
Shift the time index, using the index's frequency if available.

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

shifted : NDFrame

tz_convert (*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

tz : string or pytz.timezone object axis : the axis to convert level : int, str, default None

If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

TypeError If the axis is tz-naive.

tz_localize (*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*)
Localize tz-naive TimeSeries to target time zone.

tz : string or pytz.timezone object axis : the axis to localize level : int, str, default None

If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise']

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times

- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

TypeError If the TimeSeries is tz-aware and tz is not None.

unstack (*level=-1*, *fill_value=None*)

Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels. If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex). The level involved will automatically get sorted.

level [int, string, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name

fill_value [replace NaN with this value if the unstack produces] missing values

New in version 0.18.0.

DataFrame.pivot : Pivot a table based on column values. DataFrame.stack : Pivot a level of the column labels (inverse operation

from *unstack*).

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                    ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a   1.0
     b   2.0
two  a   3.0
     b   4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a   b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
   one  two
a  1.0   3.0
b  2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a  1.0
     b  2.0
two  a  3.0
     b  4.0
dtype: float64
```

unstacked : DataFrame or Series

update (*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

other [DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

join [{'left'}, default 'left'] Only left join is implemented, keeping the index and columns of the original object.

overwrite [bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.
- False: only update values that are NA in the original DataFrame.

filter_func [callable(1d-array) -> boolean 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

raise_conflict [bool, default False] If True, will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

ValueError When *raise_conflict* is True and there's overlapping non-NA data.

dict.update : Similar method for dictionaries. DataFrame.merge : For column(s)-on-columns(s) operations.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
1  b  e
2  c  f
```

For Series, it's name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
```

(continues on next page)

(continued from previous page)

```
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A      B
0  1    4.0
1  2  500.0
2  3    6.0
```

values

Return a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

numpy.ndarray The values of the DataFrame.

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age':    [ 3,  29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3      94      31
1   29     170     115
>>> df.dtypes
age       int64
height    int64
weight    int64
dtype: object
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('monkey', np.nan, None)],
...                    columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name          object
max_speed    float64
rank          object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
```

(continues on next page)

(continued from previous page)

```
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

pandas.DataFrame.index : Retrievie the index labels pandas.DataFrame.columns : Retrieving the column names

var(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

var : Series or DataFrame (if level specified)

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
```

(continues on next page)

(continued from previous page)

```
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

DataFrame.mask()

xs(*key*, *axis=0*, *level=None*, *drop_level=True*)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on the rows (axis=0).

key [object] Some label contained in the index, or partially in a MultiIndex

axis [int, default 0] Axis to retrieve cross-section on

level [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level [boolean, default True] If False, returns object with same levels as self.

```
>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
A    4
B    5
C    2
Name: a
>>> df.xs('C', axis=1)
a    2
b    9
c    3
Name: C
```

```
>>> df
                     A  B  C  D
first second third
bar   one    1       4  1  8  9
      two    1       7  5  5  0
baz   one    1       6  6  8  0
      three  2       5  3  5  3
>>> df.xs(('baz', 'three'))
       A  B  C  D
third
2      5  3  5  3
>>> df.xs('one', level=1)
             A  B  C  D
first third
bar   1      4  1  8  9
baz   1      6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
        A  B  C  D
```

(continues on next page)

(continued from previous page)

```
second
three    5  3  5  3
```

xs : Series or DataFrame

xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of xs functionality, see MultiIndex Slicers

class Fred2.Core.Result.**TAPPredictionResult**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

Bases: *Fred2.Core.Result.AResult*

A *TAPPredictionResult* object is a `pandas.DataFrame` with single-indexing, where column Ids are the ‘ prediction names of the different prediction methods, and row ID the *Peptide* object

TAPPredictionResult:

Peptide Obj	Method Name
Peptide1	-15.34
Peptide2	23.34

T

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
                0     1
name        Alice   Bob
score         9.5     8
employed    False  True
kids            0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name          object
score        float64
employed        bool
kids           int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

abs()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

abs Series/DataFrame containing the absolute value of each element.

For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
```

(continues on next page)

(continued from previous page)

```
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
    a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
    a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

numpy.absolute : calculate the absolute value element-wise.

add(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.radd

add_prefix(*prefix*)

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

prefix [str] The string to add before each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_suffix: Suffix row labels with string *suffix*. DataFrame.add_suffix: Suffix column labels with string *suffix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
```

(continues on next page)

(continued from previous page)

```
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
     col_A  col_B
0       1       3
1       2       4
2       3       5
3       4       6
```

add_suffix(*suffix*)

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

suffix [str] The string to add after each label.

Series or DataFrame New Series or DataFrame with updated labels.

Series.add_prefix: Prefix row labels with string *prefix*. DataFrame.add_prefix: Prefix column labels with string *prefix*.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4],  'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
     A_col  B_col
0       1       3
1       2       4
2       3       5
3       4       6
```

agg(*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

aggregate (*func*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

func [function, string, dictionary, or list of string/functions] Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

***args** Positional arguments to pass to *func*.

****kwargs** Keyword arguments to pass to *func*.

aggregated : DataFrame

agg is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

agg is an alias for *aggregate*. Use the alias.

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
max   NaN  8.0
min   1.0  2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

DataFrame.apply : Perform any type of operations. DataFrame.transform : Perform transformation type operations. pandas.core.groupby.GroupBy : Perform operations over groups. pandas.core.resample.Resampler : Perform operations over resampled bins. pandas.core.window.Rolling : Perform operations over rolling window. pandas.core.window.Expanding : Perform operations over expanding window. pandas.core.window.EWM : Perform operation over exponential weighted

window.

align(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)

Align two objects on their axes with the specified join method for each axis Index

other : DataFrame or Series join : {'outer', 'inner', 'left', 'right'}, default 'outer' axis : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None)

level [int or level name, default None] Broadcast across a level, matching Index values on the passed MultiIndex level

copy [boolean, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

method : str, default None limit : int, default None fill_axis : {0 or 'index', 1 or 'columns'}, default 0

Filling axis, method and limit

broadcast_axis [{0 or 'index', 1 or 'columns'}, default None] Broadcast values along this axis, if aligning two objects of different dimensions

(left, right) [(DataFrame, type of other)] Aligned objects

all (*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether all elements are True, potentially over an axis.

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

all : Series or DataFrame (if level specified)

pandas.Series.all : Return True if all elements are True pandas.DataFrame.any : Return True if one (or more) elements are True

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

any(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

axis [{0 or 'index', 1 or 'columns', None}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

bool_only [boolean, default None] Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

****kwargs** [any, default None] Additional keywords have no effect but might be accepted for compatibility with NumPy.

any : Series or DataFrame (if level specified)

pandas.DataFrame.all : Return whether all elements are True.

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([True, False]).any()
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
```

(continues on next page)

(continued from previous page)

```
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

append(*other*, *ignore_index=False*, *verify_integrity=False*, *sort=None*)

Append rows of *other* to the end of this frame, returning a new object. Columns not in this frame are added as new columns.

other [DataFrame or Series/dict-like object, or list of these] The data to append.

ignore_index [boolean, default False] If True, do not use the index labels.

verify_integrity [boolean, default False] If True, raise ValueError on creating index with duplicates.

sort [boolean, default None] Sort columns if the columns of *self* and *other* are not aligned. The default sorting is deprecated and will change to not-sorting in a future version of pandas. Explicitly pass `sort=True` to silence the warning and sort. Explicitly pass `sort=False` to silence the warning and not sort.

New in version 0.23.0.

appended : DataFrame

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

pandas.concat [General function to concatenate DataFrame, Series] or Panel objects

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With *ignore_index* set to True:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
...     df = df.append({'A': i}, ignore_index=True)
>>> df
   A
0  0
1  1
2  2
3  3
4  4
```

More efficient:

```
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
...           ignore_index=True)
   A
0  0
1  1
2  2
3  3
4  4
```

apply (*func*, *axis=0*, *broadcast=None*, *raw=False*, *reduce=None*, *result_type=None*, *args=()*, ***kwds*)
Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

func [function] Function to apply to each column or row.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

broadcast [bool, optional] Only relevant for aggregation functions:

- `False` or `None` : returns a Series whose length is the length of the index or the number of columns (based on the *axis* parameter)
- `True` : results will be broadcast to the original shape of the frame, the original index and columns will be retained.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by result_type='broadcast'.

raw [bool, default False]

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

reduce [bool or None, default None] Try to apply reduction procedures. If the DataFrame is empty, *apply* will use *reduce* to determine whether the result should be a Series or a DataFrame. If `reduce=None` (the default), *apply*'s return value will be guessed by calling *func* on an empty Series (note: while guessing, exceptions raised by *func* will be ignored). If `reduce=True` a Series will always be returned, and if `reduce=False` a DataFrame will always be returned.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by `result_type='reduce'`.

result_type [{'expand', 'reduce', 'broadcast', None}, default None] These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.
- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

New in version 0.23.0.

args [tuple] Positional arguments to pass to *func* in addition to the array/series.

****kwds** Additional keyword arguments to pass as keywords arguments to *func*.

In the current implementation apply calls *func* twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if *func* has side-effects, as they will take effect twice for the first column/row.

DataFrame.applymap: For elementwise operations DataFrame.aggregate: only perform aggregating type operations DataFrame.transform: only perform transformating type operations

```
>>> df = pd.DataFrame([[4, 9],] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
```

(continues on next page)

(continued from previous page)

```
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Retuning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing result_type='expand' will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
```

(continues on next page)

(continued from previous page)

```
1  1  2
2  1  2
```

applied : Series or DataFrame

applymap (*func*)

Apply a function to a Dataframe elementwise.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

func [callable] Python function, returns a single value from a single value.

DataFrame Transformed DataFrame.

DataFrame.apply : Apply a function along input axis of DataFrame

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
       0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.applymap(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.applymap(lambda x: x**2)
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

But it's better to avoid applymap in that case.

```
>>> df ** 2
           0          1
0   1.000000   4.494400
1  11.262736  20.857489
```

as_blocks (*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in as_matrix)

copy : boolean, default True

values : a dict of dtype -> Constructor Types

as_matrix (*columns=None*)

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

columns: list, optional, default:None If None, return all columns, otherwise, returns specified columns.

values [ndarray] If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By numpy.find_common_type convention, mixing int64 and uint64 will result in a flot64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use '.values'.

pandas.DataFrame.values

asfreq(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)
Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

freq : DateOffset object, or string method : {'backfill'/'bfill', 'pad'/'ffill'}, default None

Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill

how [{'start', 'end'}, default end] For PeriodIndex only, see PeriodIndex.asfreq

normalize [bool, default False] Whether to reset output index to midnight

fill_value: scalar, optional Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

converted : type of caller

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
                     s
2000-01-01 00:00:00    0.0
2000-01-01 00:01:00    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:03:00    3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                     s
2000-01-01 00:00:00    0.0
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                       s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    2.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    3.0
2000-01-01 00:03:00    3.0
```

reindex

To learn more about the frequency strings, please see this link.

asof (*where*, *subset=None*)

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

where : date or array of dates subset : string or list of strings, default None

if not None use these columns for NaN propagation

Dates are assumed to be sorted Raises if this is not the case

where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

where is Index: same shape object as input

merge_asof

assign (***kwargs*)

Assign new columns to a DataFrame, returning a new object (a copy) with the new columns added to the original ones. Existing columns that are re-assigned will be overwritten.

kwargs [keyword, value pairs] keywords are the column names. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

df [DataFrame] A new DataFrame with the new columns in addition to all the existing columns.

Assigning multiple columns within the same `assign` is possible. For Python 3.6 and above, later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order. For Python 3.5 and below, the order of keyword arguments is not specified, you cannot refer to newly created or modified columns. All items are computed first, and then assigned in alphabetical order.

Changed in version 0.23.0: Keyword argument order is maintained for Python 3.6 and later.

```
>>> df = pd.DataFrame({'A': range(1, 11), 'B': np.random.randn(10)})
```

Where the value is a callable, evaluated on *df*:

```
>>> df.assign(ln_A = lambda x: np.log(x.A))
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the value already exists and is inserted:

```
>>> newcol = np.log(df['A'])
>>> df.assign(ln_A=newcol)
    A         B      ln_A
0   1  0.426905  0.000000
1   2 -0.780949  0.693147
2   3 -0.418711  1.098612
3   4 -0.269708  1.386294
4   5 -0.274002  1.609438
5   6 -0.500792  1.791759
6   7  1.649697  1.945910
7   8 -1.495604  2.079442
8   9  0.549296  2.197225
9  10 -0.758542  2.302585
```

Where the keyword arguments depend on each other

```
>>> df = pd.DataFrame({'A': [1, 2, 3]})
```

```
>>> df.assign(B=df.A, C=lambda x:x['A']+ x['B'])
    A  B  C
 0  1  1  2
 1  2  2  4
 2  3  3  6
```

astype(***kwargs*)

Cast a pandas object to a specified dtype `dtype`.

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

copy [bool, default True.] Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

errors [{'raise', 'ignore'}, default 'raise'.] Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

raise_on_error [raise on invalid input] Deprecated since version 0.20.0: Use `errors` instead

kwargs : keyword arguments to pass on to the constructor

casted : type of caller

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> ser.astype('category', ordered=True, categories=[2, 1])
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to a numeric type. numpy.ndarray.astype : Cast a numpy array to a specified type.

at

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.iat [Access a single value for a row/column pair by integer] position

DataFrame.loc : Access a group of rows and columns by label(s) Series.at : Access a single value using a label

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

KeyError When label does not exist in DataFrame

at_time (*time*, *asof=False*)

Select values at particular time of day (e.g. 9:30AM).

TypeError If the index is not a `DatetimeIndex`

time : datetime.time or string

values_at_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

between_time : Select values between particular times of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_at_time : Get just the index locations for

values at particular time of the day

axes

Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['col1', 'col2'],
dtype='object')]
```

between_time (*start_time*, *end_time*, *include_start=True*, *include_end=True*)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting `start_time` to be later than `end_time`, you can get the times that are *not* between the two times.

TypeError If the index is not a `DatetimeIndex`

start_time : datetime.time or string end_time : datetime.time or string include_start : boolean, default True include_end : boolean, default True

values_between_time : type of caller

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
                     A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> ts.between_time('0:15', '0:45')
                     A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

at_time : Select values at a particular time of the day first : Select initial periods of time series based on a date offset last : Select final periods of time series based on a date offset DatetimeIndex.indexer_between_time : Get just the index locations for

values between particular times of the day

bfill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='bfill')`

blocks
Internal property, property synonym for as_blocks()

Deprecated since version 0.21.0.

bool()
Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

boxplot(*column=None*, *by=None*, *ax=None*, *fontsize=None*, *rot=0*, *grid=True*, *figsize=None*, *layout=None*, *return_type=None*, ***kwds*)
Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to *1.5 * IQR (IQR = Q3 - Q1)* from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia's entry for boxplot.

column [str or list of str, optional] Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.

by [str or array-like, optional] Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in *by*.

ax [object of class matplotlib.axes.Axes, optional] The matplotlib axes to be used by boxplot.

fontsize [float or str] Tick label font size in points or as a string (e.g., *large*).

rot [int or float, default 0] The rotation angle of labels (in degrees) with respect to the screen coordinate sytem.

grid [boolean, default True] Setting this to True will show the grid.

figsize [A tuple (width, height) in inches] The size of the figure to create in matplotlib.

layout [tuple (rows, columns), optional] For example, (3, 5) will display the subplots using 3 columns and 5 rows, starting from the top-left.

return_type [{'axes', 'dict', 'both'} or None, default 'axes'] The kind of object to return. The default is `axes`.

- 'axes' returns the matplotlib axes the boxplot is drawn on.
- 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot.
- 'both' returns a namedtuple with the axes and dict.
- when grouping with `by`, a Series mapping columns to `return_type` is returned.

  If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

result :

The return type depends on the *return_type* parameter:

- 'axes' : object of class matplotlib.axes.Axes

- 'dict' : dict of matplotlib.lines.Line2D objects
- 'both' : a nametuple with strucure (ax, lines)

For data grouped with `by`:

- `Series`
- `array` (for `return_type = None`)

Series.plot.hist: Make a histogram. matplotlib.pyplot.boxplot : Matplotlib equivalent plot.

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`. For instance:

A list of strings (i.e. `['X', 'Y']`) can be passed to boxplot in order to group the data by combination of the variables in the x-axis:

The layout of boxplot can be adjusted giving a tuple to `layout`:

Additional formatting can be done to the boxplot, like suppressing the grid (`grid=False`), rotating the labels in the x-axis (i.e. `rot=45`) or changing the fontsize (i.e. `fontsize=15`):

The parameter `return_type` can be used to select the type of element returned by *boxplot*. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```
>>> boxplot = df.boxplot(column=['Col1','Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

When grouping with `by`, a Series mapping columns to `return_type` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                      return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is *None*, a NumPy array of axes with the same shape as `layout` is returned:

```
>>> boxplot =  df.boxplot(column=['Col1', 'Col2'], by='X',
...                       return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

clip (*lower=None*, *upper=None*, *axis=None*, *inplace=False*, **args*, ***kwargs*)
Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

lower [float or array_like, default None] Minimum threshold value. All values below this threshold will be set to it.

upper [float or array_like, default None] Maximum threshold value. All values above this threshold will be set to it.

axis [int or string axis name, optional] Align object with lower and upper along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

***args, ****kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

clip_lower : Clip values below specified threshold(s). clip_upper : Clip values above specified threshold(s).

Series or DataFrame Same type as calling object with the values outside the clip boundaries replaced

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9     -2
1     -3     -7
2      0      6
3     -1      8
4      5     -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6     -2
1     -3     -4
2      0      6
3     -1      6
4      5     -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

clip_lower(*threshold*, *axis=None*, *inplace=False*)
Return copy of the input with values below a threshold truncated.

threshold [numeric or array-like] Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for `axis=None`, or 1-D and the same length as the axis being compared.

axis [{0 or 'index', 1 or 'columns'}, default 0] Align *self* with *threshold* along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data.

New in version 0.21.0.

Series.clip [Return copy of input with values below and above] thresholds truncated.

Series.clip_upper [Return copy of input with values above] threshold truncated.

clipped : same type as input

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0    8
1    8
2    8
3    8
4    9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0    5
1    8
2    7
3    8
4    9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

clip_upper (*threshold*, *axis=None*, *inplace=False*)

Return copy of input with values above given value(s) truncated.

threshold : float or array_like axis : int or string axis name, optional

Align object with threshold along the given axis.

inplace [boolean, default False] Whether to perform the operation in place on the data

New in version 0.21.0.

clip

clipped : same type as input

columns

The column labels of the DataFrame.

combine (*other*, *func*, *fill_value=None*, *overwrite=True*)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

other : DataFrame func : function

Function that takes two series as inputs and return a Series or a scalar

fill_value : scalar value overwrite : boolean, default True

If True then overwrite values for common keys in the calling frame

result : DataFrame

```
>>> df1 = DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, lambda s1, s2: s1 if s1.sum() < s2.sum() else s2)
   A  B
0  0  3
1  0  3
```

DataFrame.combine_first [Combine two DataFrame objects and default to] non-null values in frame calling the method

combine_first (*other*)

Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

other : DataFrame

combined : DataFrame

df1's values prioritized, use values from df2 to fill holes:

```
>>> df1 = pd.DataFrame([[1, np.nan]])
>>> df2 = pd.DataFrame([[3, 4]])
>>> df1.combine_first(df2)
   0    1
0  1  4.0
```

DataFrame.combine [Perform series-wise operation on two DataFrames] using a given function

compound(*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

compounded : Series or DataFrame (if level specified)

consolidate(*inplace=False*)

Compute NDFrame with "consolidated" internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

convert_objects(*convert_dates=True*, *convert_numeric=False*, *convert_timedeltas=True*, *copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

convert_dates [boolean, default True] If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

convert_numeric [boolean, default False] If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

convert_timedeltas [boolean, default True] If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

copy [boolean, default True] If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Return a fixed frequency timedelta index,

with day as the default.

converted : same as input object

copy(*deep=True*)

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

deep [bool, default True] Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

copy [Series, DataFrame or Panel] Object type matches caller.

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
```

(continues on next page)

(continued from previous page)

```
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

corr (*method='pearson'*, *min_periods=1*)

Compute pairwise correlation of columns, excluding NA/null values

method [{'pearson', 'kendall', 'spearman'}]

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation

y : DataFrame

corrwith (*other*, *axis=0*, *drop=False*)

Compute pairwise correlation between rows or columns of two DataFrame objects.

other : DataFrame, Series axis : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' to compute column-wise, 1 or 'columns' for row-wise

drop [boolean, default False] Drop missing indices from result, default returns union of all

correls : Series

count (*axis=0*, *level=None*, *numeric_only=False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, and optionally *numpy.inf* (depending on *pandas.options.mode.use_inf_as_na*) are considered NA.

axis [{0 or 'index', 1 or 'columns'}, default 0] If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each **row**.

level [int or str, optional] If the axis is a *MultiIndex* (hierarchical), count along a particular *level*, collapsing into a *DataFrame*. A *str* specifies the level name.

numeric_only [boolean, default False] Include only *float*, *int* or *boolean* data.

Series or DataFrame For each column/row the number of non-NA/null entries. If *level* is specified returns a *DataFrame*.

Series.count: number of non-NA elements in a Series DataFrame.shape: number of DataFrame rows and columns (including NA

elements)

DataFrame.isna: boolean same-sized DataFrame showing places of NA elements

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                    ["John", "Myla", None, "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> df
   Person   Age  Single
0    John  24.0   False
1    Myla   NaN    True
2    None  21.0    True
3    John  33.0    True
4    Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    4
Age       4
Single    5
dtype: int64
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    2
3    3
4    3
dtype: int64
```

Counts for one level of a *MultiIndex*:

```
>>> df.set_index(["Person", "Single"]).count(level="Person")
        Age
Person
John      2
Myla      1
```

cov(*min_periods=None*)

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the covariance matrix of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as `NaN`.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

min_periods [int, optional] Minimum number of observations required per pair of columns to have a valid result.

DataFrame The covariance matrix of the series of the DataFrame.

pandas.Series.cov : compute covariance with another Series pandas.core.window.EWM.cov: expoential weighted sample covariance pandas.core.window.Expanding.cov : expanding sample covariance pandas.core.window.Rolling.cov : rolling sample covariance

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1.

For DataFrames that have Series that are missing data (assuming that data is missing at random) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See Estimation of covariance matrices for more details.

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                   columns=['dogs', 'cats'])
>>> df.cov()
          dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
          a         b         c         d         e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                   columns=['a', 'b', 'c'])
```

(continues on next page)

(continued from previous page)

```
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
          a         b         c
a  0.316741       NaN -0.150812
b       NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

cummax (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummax : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.max [Similar functionality] but ignores `NaN` values.

DataFrame.max [Return the maximum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cummin (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cummin : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
```

(continues on next page)

(continued from previous page)

```
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.min [Similar functionality] but ignores `NaN` values.

DataFrame.min [Return the minimum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumprod (*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumprod : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

pandas.core.window.Expanding.prod [Similar functionality] but ignores `NaN` values.

DataFrame.prod [Return the product over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

cumsum(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis. 0 is equivalent to None or 'index'.

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

***args, **kwargs :** Additional keywords have no effect but might be accepted for compatibility with NumPy.

cumsum : Series or DataFrame

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

pandas.core.window.Expanding.sum [Similar functionality] but ignores `NaN` values.

DataFrame.sum [Return the sum over] DataFrame axis.

DataFrame.cummax : Return cumulative maximum over DataFrame axis. DataFrame.cummin : Return cumulative minimum over DataFrame axis. DataFrame.cumsum : Return cumulative sum over DataFrame axis. DataFrame.cumprod : Return cumulative product over DataFrame axis.

describe (*percentiles=None*, *include=None*, *exclude=None*)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

percentiles [list-like of numbers, optional] The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

include ['all', list-like of dtypes or None (default), optional] A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
- None (default) : The result will include all numeric columns.

exclude [list-like of dtypes or None (default), optional,] A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
- None (default) : The result will exclude nothing.

summary: Series/DataFrame of summary statistics

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
... ])
>>> s.describe()
count                       3
unique                      2
top       2010-01-01 00:00:00
freq                        2
first     2000-01-01 00:00:00
last      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                     'numeric': [1, 2, 3],
...                     'categorical': pd.Categorical(['d','e','f'])
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
        categorical  numeric object
count             3      3.0      3
unique            3      NaN      3
top               f      NaN      c
freq              1      NaN      1
mean            NaN      2.0    NaN
std             NaN      1.0    NaN
min             NaN      1.0    NaN
25%             NaN      1.5    NaN
50%             NaN      2.0    NaN
75%             NaN      2.5    NaN
max             NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[np.object])
       object
count       3
unique      3
top         c
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              f
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      c
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
        categorical  numeric
count            3      3.0
```

(continues on next page)

(continued from previous page)

```
unique            3      NaN
top               f      NaN
freq              1      NaN
mean            NaN      2.0
std             NaN      1.0
min             NaN      1.0
25%             NaN      1.5
50%             NaN      2.0
75%             NaN      2.5
max             NaN      3.0
```

DataFrame.count DataFrame.max DataFrame.min DataFrame.mean DataFrame.std DataFrame.select_dtypes

diff (*periods=1*, *axis=0*)

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

periods [int, default 1] Periods to shift for calculating difference, accepts negative values.

axis [{0 or 'index', 1 or 'columns'}, default 0] Take difference over rows (0) or columns (1).

New in version 0.16.1..

diffed : DataFrame

Series.diff: First discrete difference for a Series. DataFrame.pct_change: Percent change over given number of periods. DataFrame.shift: Shift index by desired number of periods with an

optional time freq.

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
3  4  3  16
4  5  5  25
5  6  8  36
```

```
>>> df.diff()
     a    b     c
0  NaN  NaN   NaN
1  1.0  0.0   3.0
2  1.0  1.0   5.0
3  1.0  1.0   7.0
4  1.0  2.0   9.0
5  1.0  3.0  11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
    a    b     c
0 NaN  0.0   0.0
1 NaN -1.0   3.0
2 NaN -1.0   7.0
3 NaN -1.0  13.0
4 NaN  0.0  20.0
5 NaN  2.0  28.0
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
     a    b     c
0  NaN  NaN   NaN
1  NaN  NaN   NaN
2  NaN  NaN   NaN
3  3.0  2.0  15.0
4  3.0  4.0  21.0
5  3.0  6.0  27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
     a    b     c
0 -1.0  0.0  -3.0
1 -1.0 -1.0  -5.0
2 -1.0 -1.0  -7.0
3 -1.0 -2.0  -9.0
4 -1.0 -3.0 -11.0
5  NaN  NaN   NaN
```

div(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

divide(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

dot(*other*)

Matrix multiplication with DataFrame or Series objects. Can also be called using *self @ other* in Python >= 3.5.

other : DataFrame or Series

dot_product : DataFrame or Series

drop(*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

labels [single label or list-like] Index or column labels to drop.

axis [{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index, columns [single label or list-like] Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).

New in version 0.21.0.

level [int or level name, optional] For MultiIndex, level from which the labels will be removed.

inplace [bool, default False] If True, do operation inplace and return None.

errors [{'ignore', 'raise'}, default 'raise'] If 'ignore', suppress error and only existing labels are dropped.

dropped : pandas.DataFrame

DataFrame.loc : Label-location based indexer for selection by label. DataFrame.dropna : Return DataFrame with labels on given axis omitted

where (all or any) data are missing

DataFrame.drop_duplicates [Return DataFrame with duplicate rows] removed, optionally only considering certain columns

Series.drop : Return Series with specified index labels removed.

KeyError If none of the labels are found in the selected axis

```
>>> df = pd.DataFrame(np.arange(12).reshape(3,4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                              [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3,0.2]])
>>> df
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
lama    speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
```

(continues on next page)

(continued from previous page)

```
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

drop_duplicates (*subset=None*, *keep='first'*, *inplace=False*)

Return DataFrame with duplicate rows removed, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Drop duplicates except for the first occurrence.
- `last` : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

inplace [boolean, default False] Whether to drop duplicates in place or to return a copy

deduplicated : DataFrame

dropna (*axis=0*, *how='any'*, *thresh=None*, *subset=None*, *inplace=False*)

Remove missing values.

See the User Guide for more on which values are considered missing, and how to work with missing data.

axis [{0 or 'index', 1 or 'columns'}, default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Deprecated since version 0.23.0:: Pass tuple or list to drop on multiple

axes.

how [{'any', 'all'}, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

thresh [int, optional] Require that many non-NA values.

subset [array-like, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

inplace [bool, default False] If True, do operation inplace and return None.

DataFrame DataFrame with NA entries dropped from it.

DataFrame.isna: Indicate missing values. DataFrame.notna : Indicate existing (non-missing) values. DataFrame.fillna : Replace missing values. Series.dropna : Drop missing values. Index.dropna : Drop missing indices.

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'born'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type of each column.

pandas.DataFrame.ftypes : dtype and sparsity information.

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
string              object
dtype: object
```

duplicated(*subset=None*, *keep='first'*)

Return boolean Series denoting duplicate rows, optionally only considering certain columns

subset [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

keep [{'first', 'last', False}, default 'first']

- `first` : Mark duplicates as `True` except for the first occurrence.
- `last` : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

duplicated : Series

empty

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

bool If DataFrame is empty, return True, if not return False.

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
```

(continues on next page)

(continued from previous page)

```
>>> df.dropna().empty
True
```

pandas.Series.dropna pandas.DataFrame.dropna

eq(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods eq

equals(*other*)
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

eval(*expr*, *inplace=False*, ***kwargs*)
Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

expr [str] The expression string to evaluate.

inplace [bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

New in version 0.18.0..

kwargs [dict] See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

ndarray, scalar, or pandas object The result of the evaluation.

DataFrame.query [Evaluates a boolean expression to query the columns] of a frame.

DataFrame.assign [Can evaluate an expression or function to create new] values for a column.

pandas.eval [Evaluate a Python expression as a string using various] backends.

For more details see the API documentation for `eval()`. For detailed examples see enhancing performance with eval.

```
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
>>> df.eval('A + B')
0    11
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
>>> df
   A   B
0  1  10
1  2   8
2  3   6
3  4   4
4  5   2
```

Use `inplace=True` to modify the original DataFrame.

```
>>> df.eval('C = A + B', inplace=True)
>>> df
   A   B   C
0  1  10  11
1  2   8  10
2  3   6   9
3  4   4   8
4  5   2   7
```

ewm(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*)
Provides exponential weighted functions

New in version 0.18.0.

com [float, optional] Specify decay in terms of center of mass, $\alpha = 1/(1 + com),$ for $com \geq 0$

span [float, optional] Specify decay in terms of span, $\alpha = 2/(span + 1),$ for $span \geq 1$

halflife [float, optional] Specify decay in terms of half-life, $\alpha = 1 - exp(log(0.5)/halflife),$ for $halflife > 0$

alpha [float, optional] Specify smoothing factor α directly, $0 < \alpha \leq 1$

New in version 0.18.0.

min_periods [int, default 0] Minimum number of observations in window required to have a value (otherwise result is NA).

adjust [boolean, default True] Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

ignore_na [boolean, default False] Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights (1-alpha)**(n-1), (1-alpha)**(n-2), . . . , 1-alpha, 1.

When adjust is False, weighted averages are calculated recursively as: weighted_average[0] = arg[0]; weighted_average[i] = (1-alpha)*weighted_average[i-1] + alpha*arg[i].

When ignore_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are (1-alpha)**2 and 1 (if adjust is True), and (1-alpha)**2 and alpha (if adjust is False).

When ignore_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are 1-alpha and 1 (if adjust is True), and 1-alpha and alpha (if adjust is False).

More details can be found at http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows

rolling : Provides rolling window calculations expanding : Provides expanding transformations.

expanding(*min_periods=1*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

min_periods [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

center [boolean, default False] Set the labels at the center of the window.

axis : int or string, default 0

a Window sub-classed for the particular operation

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

rolling : Provides rolling window calculations ewm : Provides exponential weighted functions

ffill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna(method='ffill')`

fillna(*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*, *downcast=None*, ***kwargs*)
Fill NA/NaN values using the specified method

value [scalar, dict, Series, or DataFrame] Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

method [{'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

axis : {0 or 'index', 1 or 'columns'} inplace : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

limit [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

downcast [dict, default is None] a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

interpolate : Fill NaN values using interpolation. reindex, asfreq

filled : DataFrame

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                   columns=list('ABCD'))
>>> df
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
3  NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
    A   B   C   D
0   0.0 2.0 0.0 0
1   3.0 4.0 0.0 1
2   0.0 0.0 0.0 5
3   0.0 3.0 0.0 4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
    A   B   C   D
0   NaN 2.0 NaN 0
1   3.0 4.0 NaN 1
2   3.0 4.0 NaN 5
3   3.0 3.0 NaN 4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 2.0 1
2   0.0 1.0 2.0 5
3   0.0 3.0 2.0 4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
    A   B   C   D
0   0.0 2.0 2.0 0
1   3.0 4.0 NaN 1
2   NaN 1.0 NaN 5
3   NaN 3.0 NaN 4
```

filter (*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

items [list-like] List of info axis to restrict to (must not all be present)

like [string] Keep info axis where "arg in col == True"

regex [string (regular expression)] Keep info axis with re.search(regex, col) == True

axis [int or string axis name] The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame

same type as input object

```
>>> df
one  two  three
mouse     1    2      3
rabbit    4    5      6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
one  two  three
rabbit    4    5      6
```

pandas.DataFrame.loc

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

filter_result(*expressions*)

Filters a result data frame based on a specified expression consisting of a list of triple with (method_name, comparator, threshold). The expression is applied to each row. If any of the columns fulfill the criteria the row remains.

Parameters **expressions** (*list((str, comparator, float))*) – A list of triples consisting of (method_name, comparator, threshold)

Returns A new filtered result object

Return type *TAPPredictionResult*

first(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
            A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calender days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

subset : type of caller

last : Select final periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

first_valid_index()

Return index for first non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

floordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rfloordiv

classmethod from_csv(*path*, *header=0*, *sep=','*, *','*, *index_col=0*, *parse_dates=True*, *encoding=None*, *tupleize_cols=None*, *infer_datetime_format=False*)
Read CSV file.

Deprecated since version 0.21.0: Use `pandas.read_csv()` instead.

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a DataFrame of time series data.

This method only differs from the preferred `pandas.read_csv()` in some defaults:

- *index_col* is `0` instead of `None` (take first column as index by default)
- *parse_dates* is `True` instead of `False` (try parsing the index as datetime by default)

So a `pd.DataFrame.from_csv(path)` can be replaced by `pd.read_csv(path, index_col=0, parse_dates=True)`.

path : string file path or file handle / StringIO header : int, default 0

Row to use as header (skip prior rows)

sep [string, default ','] Field delimiter

index_col [int or sequence, default 0] Column to use for index. If a sequence is given, a MultiIndex is used. Different default from read_table

parse_dates [boolean, default True] Parse dates. Different default from read_table

tupleize_cols [boolean, default False] write multi_index columns as a list of tuples (if True) or new (expanded format) if False)

infer_datetime_format: boolean, default False If True and *parse_dates* is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

pandas.read_csv

y : DataFrame

classmethod from_dict(*data*, *orient='columns'*, *dtype=None*, *columns=None*)
Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

data [dict] Of the form {field : array-like} or {field : dict}.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'.

dtype [dtype, default None] Data type to force, otherwise infer.

columns [list, default None] Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'`.

New in version 0.23.0.

pandas.DataFrame

DataFrame.from_records [DataFrame from ndarray (structured] dtype), list of tuples, dict, or DataFrame

DataFrame : DataFrame object creation using constructor

By default the keys of the dict become the DataFrame columns:

```
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
```

Specify `orient='index'` to create the DataFrame using dictionary keys as rows:

```
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
       0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                        columns=['A', 'B', 'C', 'D'])
       A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
```

classmethod from_items(*items*, *columns=None*, *orient='columns'*)
Construct a dataframe from a list of tuples

Deprecated since version 0.23.0: *from_items* is deprecated and will be removed in a future version. Use `DataFrame.from_dict(dict(items))` instead. `DataFrame.from_dict(OrderedDict(items))` may be used to preserve the key order.

Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the columns, but depends on the specified orientation). The values should be arrays or Series.

items [sequence of (key, value) pairs] Values should be arrays or Series.

columns [sequence of column labels, optional] Must be passed if orient='index'.

orient [{'columns', 'index'}, default 'columns'] The "orientation" of the data. If the keys of the input correspond to column labels, pass 'columns' (default). Otherwise if the keys correspond to the index, pass 'index'.

frame : DataFrame

classmethod from_records(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float=False*, *nrows=None*)

Convert structured or record ndarray to DataFrame

data : ndarray (structured dtype), list of tuples, dict, or DataFrame index : string, list of fields, array-like

Field of array to use as the index, alternately a specific set of input labels to use

exclude [sequence, default None] Columns or fields to exclude

columns [sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

coerce_float [boolean, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

df : DataFrame

ftypes

Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See the User Guide for more.

pandas.Series The data type and indication of sparse/dense of each column.

pandas.DataFrame.dtypes: Series with just dtype information. pandas.SparseDataFrame : Container for sparse tabular data.

Sparse data should have the same dtypes as its dense representation.

```
>>> import numpy as np
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).ftypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).ftypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

ge(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods ge

get (*key*, *default=None*)
Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

key : object

value : type of items contained in object

get_dtype_counts ()
Return counts of unique dtypes in this object.

dtype [Series] Series with the count of columns with each dtype.

dtypes : Return the dtypes in this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

get_ftype_counts ()
Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

dtype [Series] Series with the count of columns with each type and sparsity (dense/sparse)

ftypes [Return ftypes (indication of sparse/dense and dtype) in] this object.

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
  str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

get_value (*index*, *col*, *takeable=False*)
Quickly retrieve single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label takeable : interpret the index/col as indexers, default False

value : scalar value

get_values()

Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a *pandas.SparseArray*, the data are first converted to a dense representation.

numpy.ndarray Numpy representation of DataFrame

values : Numpy representation of DataFrame. pandas.SparseArray : Container for sparse data.

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a      b    c
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
     a    c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

groupby(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=False*, *observed=False*, ***kwargs*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

by [mapping, function, label, or list of labels] Used to determine the groups for the groupby. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted a (single) key.

axis : int, default 0 level : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

as_index [boolean, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output

sort [boolean, default True] Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.

group_keys [boolean, default True] When calling apply, add group keys to index to identify pieces

squeeze [boolean, default False] reduce the dimensionality of the return type if possible, otherwise return a consistent type

observed [boolean, default False] This only applies if any of the groupers are Categoricals If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

GroupBy object

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

See the user guide for more.

resample [Convenience method for frequency conversion and resampling] of time series.

gt (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods gt

head (*n=5*)
Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

n [int, default 5] Number of rows to select.

obj_head [type of caller] The first *n* rows of the caller object.

pandas.DataFrame.tail: Returns the last *n* rows.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
```

(continues on next page)

(continued from previous page)

```
3       lion
4     monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

hist(*column=None*, *by=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *ax=None*, *sharex=False*, *sharey=False*, *figsize=None*, *layout=None*, *bins=10*, ***kwds*)
Make a histogram of the DataFrame's.

A histogram is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

data [DataFrame] The pandas object holding the data.

column [string or sequence] If passed, will be used to limit data to a subset of columns.

by [object, optional] If passed, then used to form histograms for separate groups.

grid [boolean, default True] Whether to show axis grid lines.

xlabelsize [int, default None] If specified changes the x-axis label size.

xrot [float, default None] Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

ylabelsize [int, default None] If specified changes the y-axis label size.

yrot [float, default None] Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

ax [Matplotlib axes object, default None] The axes to plot the histogram on.

sharex [boolean, default True if ax is None else False] In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

sharey [boolean, default False] In case subplots=True, share y axis and set some y axis labels to invisible.

figsize [tuple] The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

layout [tuple, optional] Tuple of (rows, columns) for the layout of the histograms.

bins [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

****kwds** All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

axes : matplotlib.AxesSubplot or numpy.ndarray of them

matplotlib.pyplot.hist : Plot a histogram using matplotlib.

iat

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

DataFrame.at : Access a single value for a row/column label pair DataFrame.loc : Access a group of rows and columns by label(s) DataFrame.iloc : Access a group of rows and columns by integer position(s)

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

IndexError When integer position is out of bounds

idxmax (*axis=0*, *skipna=True*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmax : Series

This method is the DataFrame version of `ndarray.argmax`.

Series.idxmax

idxmin (*axis=0*, *skipna=True*)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

axis [{0 or 'index', 1 or 'columns'}, default 0] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

skipna [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

ValueError

- If the row/column is empty

idxmin : Series

This method is the DataFrame version of `ndarray.argmin`.

Series.idxmin

iloc

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at Selection by Position

index

The index (row labels) of the DataFrame.

infer_objects()

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

pandas.to_datetime : Convert argument to datetime. pandas.to_timedelta : Convert argument to timedelta. pandas.to_numeric : Convert argument to numeric typeR

converted : same type as input object

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

info(*verbose=None*, *buf=None*, *max_cols=None*, *memory_usage=None*, *null_counts=None*)
Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

verbose [bool, optional] Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

buf [writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

max_cols [int, optional] When to switch from the verbose to the truncated output. If the DataFrame has more than *max_cols* columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

memory_usage [bool, str, optional] Specifies whether total memory usage of the DataFrame elements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources.

null_counts [bool, optional] Whether to show the non-null counts. By default, this is shown only if the frame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

None This method prints a summary of a DataFrame and returns None.

DataFrame.describe: Generate descriptive statistics of DataFrame columns.

DataFrame.memory_usage: Memory usage of DataFrame columns.

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                    "float_col": float_values})
>>> df
   int_col text_col  float_col
0        1    alpha       0.00
1        2     beta       0.25
2        3    gamma       0.50
3        4    delta       0.75
4        5  epsilon       1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
int_col      5 non-null int64
```

(continues on next page)

(continued from previous page)

```
text_col     5 non-null object
float_col    5 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w", encoding="utf-8") as f:
...     f.write(s)
260
```

The *memory_usage* parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 22.9+ MB
```

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 188.8 MB
```

insert (*loc*, *column*, *value*, *allow_duplicates=False*)

Insert column into DataFrame at specified location.

Raises a ValueError if *column* is already contained in the DataFrame, unless *allow_duplicates* is set to

True.

loc [int] Insertion index. Must verify 0 <= loc <= len(columns)

column [string, number, or hashable object] label of the inserted column

value : int, Series, or array-like allow_duplicates : bool, optional

interpolate(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *limit_area=None*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

method [{'linear', 'time', 'index', 'values', 'nearest', 'zero',]

'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise_polynomial', 'from_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to `scipy.interpolate.interp1d`. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. df.interpolate(method='polynomial', order=4). These use the actual numerical values of the index.
- 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the scipy documentation and tutorial documentation
- 'from_derivatives' refers to BPoly.from_derivatives which replaces 'piecewise_polynomial' interpolation method in scipy 0.18

New in version 0.18.1: Added support for the 'akima' method Added interpolate method 'from_derivatives' which replaces 'piecewise_polynomial' in scipy 0.18; backwards-compatible with scipy < 0.18

axis [{0, 1}, default 0]

- 0: fill column-by-column
- 1: fill row-by-row

limit [int, default None.] Maximum number of consecutive NaNs to fill. Must be greater than 0.

limit_direction : {'forward', 'backward', 'both'}, default 'forward' limit_area : {'inside', 'outside'}, default None

- None: (default) no fill restriction
- 'inside' Only fill NaNs surrounded by valid values (interpolate).
- 'outside' Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

inplace [bool, default False] Update the NDFrame in place if possible.

downcast [optional, 'infer' or None, defaults to None] Downcast dtypes if possible.

kwargs : keyword arguments to pass on to the interpolating function.

Series or DataFrame of same shape interpolated at the NaNs

reindex, replace, fillna

Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

is_copy

isin(*values*)

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

values [iterable, Series, DataFrame or dictionary] The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dictionary, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

DataFrame of booleans

When `values` is a list:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
       A      B
0   True   True
1  False  False
2   True  False
```

When `values` is a dict:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
       A      B
0   True  False  # Note that B didn't match the 1 here.
1  False   True
2   True   True
```

When `values` is a Series or DataFrame:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
       A      B
0   True  False
1  False  False  # Column A in `other` has a 3, but not at index 1.
2   True   True
```

isna()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

isnull()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.isnull : alias of isna DataFrame.notna : boolean inverse of isna DataFrame.dropna : omit axes labels with missing values isna : top-level isna

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

items()
Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iteritems()
Iterator over (column name, Series) pairs.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. itertuples : Iterate over DataFrame rows as namedtuples of the values.

iterrows()
Iterate over DataFrame rows as (index, Series) pairs.

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
```

(continues on next page)

(continued from previous page)

```
>>> print(df['int'].dtype)
int64
```

To preserve dtypes while iterating over the rows, it is better to use *itertuples()* which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

it [generator] A generator that iterates over the rows of the frame.

itertuples : Iterate over DataFrame rows as namedtuples of the values. iteritems : Iterate over (column name, Series) pairs.

itertuples (*index=True*, *name='Pandas'*)

Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.

index [boolean, default True] If True, return the index as the first element of the tuple.

name [string, default "Pandas"] The name of the returned namedtuples or None to return regular tuples.

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

iterrows : Iterate over DataFrame rows as (index, Series) pairs. iteritems : Iterate over (column name, Series) pairs.

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [0.1, 0.2]},
                      index=['a', 'b'])
>>> df
   col1  col2
a     1   0.1
b     2   0.2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='a', col1=1, col2=0.10000000000000001)
Pandas(Index='b', col1=2, col2=0.20000000000000001)
```

ix

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at Advanced Indexing.

join (*other*, *on=None*, *how='left'*, *lsuffix=''*, *rsuffix=''*, *sort=False*)

Join columns with other DataFrame either on index or on a key column. Efficiently Join multiple DataFrame objects by index at once by passing a list.

other [DataFrame, Series with name field set, or list of DataFrame] Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame

on [name, tuple/list of names, or array-like] Column or index level name(s) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

how [{'left', 'right', 'outer', 'inner'}, default: 'left'] How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use other frame's index
- outer: form union of calling frame's index (or column if on is specified) with other frame's index, and sort it lexicographically
- inner: form intersection of calling frame's index (or column if on is specified) with other frame's index, preserving the order of the calling's one

lsuffix [string] Suffix to use from left frame's overlapping columns

rsuffix [string] Suffix to use from right frame's overlapping columns

sort [boolean, default False] Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword)

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

Support for specifying index levels as the *on* parameter was added in version 0.23.0

```
>>> caller = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                        'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> caller
    A key
0  A0  K0
1  A1  K1
2  A2  K2
3  A3  K3
4  A4  K4
5  A5  K5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
    B key
0  B0  K0
1  B1  K1
2  B2  K2
```

Join DataFrames using their indexes.

```
>>> caller.join(other, lsuffix='_caller', rsuffix='_other')
```

```
>>>     A key_caller    B key_other
    0  A0         K0   B0        K0
    1  A1         K1   B1        K1
```

(continues on next page)

(continued from previous page)

```
    2  A2          K2   B2         K2
    3  A3          K3  NaN        NaN
    4  A4          K4  NaN        NaN
    5  A5          K5  NaN        NaN
```

If we want to join using the key columns, we need to set key to be the index in both caller and other. The joined DataFrame will have key as its index.

```
>>> caller.set_index('key').join(other.set_index('key'))
```

```
>>>      A    B
    key
    K0   A0   B0
    K1   A1   B1
    K2   A2   B2
    K3   A3  NaN
    K4   A4  NaN
    K5   A5  NaN
```

Another option to join using the key columns is to use the on parameter. DataFrame.join always uses other's index but we can use any column in the caller. This method preserves the original caller's index in the result.

```
>>> caller.join(other.set_index('key'), on='key')
```

```
>>>     A key    B
    0  A0  K0   B0
    1  A1  K1   B1
    2  A2  K2   B2
    3  A3  K3  NaN
    4  A4  K4  NaN
    5  A5  K5  NaN
```

DataFrame.merge : For column(s)-on-columns(s) operations

joined : DataFrame

keys()

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major_axis for Panel.

kurt(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

kurtosis (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

kurt : Series or DataFrame (if level specified)

last (*offset*)
Convenience method for subsetting final periods of time series data based on a date offset.

TypeError If the index is not a `DatetimeIndex`

offset : string, DateOffset, dateutil.relativedelta

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
            A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calender days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

subset : type of caller

first : Select initial periods of time series based on a date offset at_time : Select values at a particular time of the day between_time : Select values between particular times of the day

last_valid_index ()
Return index for last non-NA/null value.

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

scalar : type of index

le (*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods le

loc
Access a group of rows and columns by label(s) or a boolean array.

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. `5` or `'a'`, (note that `5` is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

> **Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- A `callable` function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at Selection by Label

DataFrame.at : Access a single value for a row/column label pair DataFrame.iloc : Access group of rows and columns by integer position(s) DataFrame.xs : Returns a cross-section (row(s) or column(s)) from the

Series/DataFrame.

Series.loc : Access group of values using labels

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=['cobra', 'viper', 'sidewinder'],
...       columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
```

(continues on next page)

(continued from previous page)

```
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

KeyError: when any items are not found

lookup(*row_labels*, *col_labels*)

Label-based "fancy indexing" function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

row_labels [sequence] The row labels to use for lookup

col_labels [sequence] The column labels to use for lookup

Akin to:

```
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

values [ndarray] The found values

lt(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods lt

mad(*axis=None*, *skipna=None*, *level=None*)
Return the mean absolute deviation of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mad : Series or DataFrame (if level specified)

mask(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is True are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

`DataFrame.where()`

max (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the maximum of the values in the object. If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

max : Series or DataFrame (if level specified)

mean (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

mean : Series or DataFrame (if level specified)

median (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

median : Series or DataFrame (if level specified)

melt (*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *col_level=None*)

"Unpivots" a DataFrame from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id_vars*), while all other columns, considered measured variables (*value_vars*), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

New in version 0.20.0.

frame : DataFrame id_vars : tuple, list, or ndarray, optional

Column(s) to use as identifier variables.

value_vars [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id_vars*.

var_name [scalar] Name to use for the 'variable' column. If None it uses `frame.columns.name` or 'variable'.

value_name [scalar, default 'value'] Name to use for the 'value' column.

col_level [int or string, optional] If columns are a MultiIndex then use this level to melt.

melt pivot_table DataFrame.pivot

```
>>> import pandas as pd
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
3  a        C      2
4  b        C      4
5  c        C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> df.melt(id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname  myValname
0  a         B          1
1  b         B          3
2  c         B          5
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A  B  C
   D  E  F
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a        B      1
1  b        B      3
2  c        B      5
```

```
>>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a          B          E      1
1      b          B          E      3
2      c          B          E      5
```

memory_usage(*index=True*, *deep=False*)

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

index [bool, default True] Specifies whether to include the memory usage of the DataFrame's index in returned Series. If `index=True` the memory usage of the index the first item in the output.

deep [bool, default False] If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

sizes [Series] A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

numpy.ndarray.nbytes [Total bytes consumed by the elements of an] ndarray.

Series.memory_usage : Bytes consumed by a Series. pandas.Categorical : Memory-efficient array for string values with

many repeated values.

DataFrame.info : Concise summary of a DataFrame.

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64  complex128 object  bool
0      1      1.0      (1+0j)      1  True
1      1      1.0      (1+0j)      1  True
2      1      1.0      (1+0j)      1  True
3      1      1.0      (1+0j)      1  True
4      1      1.0      (1+0j)      1  True
```

```
>>> df.memory_usage()
Index            80
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64         40000
float64       40000
complex128    80000
object        40000
```

(continues on next page)

(continued from previous page)

```
bool           5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index             80
int64          40000
float64        40000
complex128     80000
object        160000
bool            5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5168
```

merge(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*, *sort=False*, *suffixes=('_x', '_y')*, *copy=True*, *indicator=False*, *validate=None*)

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

right : DataFrame how : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys

on [label or list] Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

left_on [label or list, or array-like] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label or list, or array-like] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [boolean, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

right_index [boolean, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index

sort [boolean, default False] Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword)

suffixes [2-length sequence (tuple, list, ...)] Suffix to apply to overlapping column names in the left and right side, respectively

copy [boolean, default True] If False, do not copy data unnecessarily

indicator [boolean or string, default False] If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in 'left' DataFrame, "right_only" for observations whose merge key only appears in 'right' DataFrame, and "both" if the observation's merge key is found in both.

validate [string, default None] If specified, checks if merge is of specified type.

- "one_to_one" or "1:1": check if merge keys are unique in both left and right datasets.
- "one_to_many" or "1:m": check if merge keys are unique in left dataset.
- "many_to_one" or "m:1": check if merge keys are unique in right dataset.
- "many_to_many" or "m:m": allowed, but does not result in checks.

New in version 0.21.0.

Support for specifying index levels as the *on*, *left_on*, and *right_on* parameters was added in version 0.23.0

```
>>> A              >>> B
    lkey value         rkey value
0   foo  1         0   foo  5
1   bar  2         1   bar  6
2   baz  3         2   qux  7
3   foo  4         3   bar  8
```

```
>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0  foo   1        foo   5
1  foo   4        foo   5
2  bar   2        bar   6
3  bar   2        bar   8
4  baz   3        NaN   NaN
5  NaN   NaN      qux   7
```

merged [DataFrame] The output type will the be same as 'left', if it is a subclass of DataFrame.

merge_ordered merge_asof DataFrame.join

merge_results(*others*)

Merges results of type :class:'~Fred2.Core.Result.TAPPredictionResult and returns the merged result

Parameters **others** (list(*TAPPredictionResult*) or *TAPPredictionResult*) – A (list of) *TAPPredictionResult* object(s)

Returns A new merged *TAPPredictionResult* object

Return type TAPPredictionResult`

min(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

This method returns the minimum of the values in the object. If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min : Series or DataFrame (if level specified)

mod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmod

mode(*axis=0*, *numeric_only=False*)
Gets the mode(s) of each element along the axis selected. Adds a row for each mode per label, fills in gaps with nan.

Note that there could be multiple values returned for the selected axis (when more than one item share the maximum frequency), which is the reason why a dataframe is returned. If you want to impute missing values with the mode in a dataframe `df`, you can just do this: `df.fillna(df.mode().iloc[0])`

axis [{0 or 'index', 1 or 'columns'}, default 0]

- 0 or 'index' : get mode of each column
- 1 or 'columns' : get mode of each row

numeric_only [boolean, default False] if True, only apply to numeric columns

modes : DataFrame (sorted)

```
>>> df = pd.DataFrame({'A': [1, 2, 1, 2, 1, 2, 3]})
>>> df.mode()
   A
0  1
1  2
```

mul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

multiply(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rmul

ndim

Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

ndarray.ndim

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

ne(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods ne

nlargest(*n*, *columns*, *keep='first'*)

Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant.

n [int] Number of rows to return.

columns [label or list of labels] Column label(s) to order by.

keep [{'first', 'last'}, default 'first'] Where there are duplicate values:

- *first* : prioritize the first occurrence(s)
- *last* : prioritize the last occurrence(s)

DataFrame The first *n* rows ordered by the given columns in descending order.

DataFrame.nsmallest [Return the first *n* rows ordered by *columns* in] ascending order.

DataFrame.sort_values : Sort DataFrame by the values DataFrame.head : Return the first *n* rows without re-ordering.

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 10, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df
    a  b    c
0   1  a  1.0
1  10  b  2.0
2   8  d  NaN
3  10  c  3.0
4  -1  e  4.0
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column "a".

```
>>> df.nlargest(3, 'a')
    a  b    c
1  10  b  2.0
3  10  c  3.0
2   8  d  NaN
```

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'a', keep='last')
    a  b    c
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

To order by the largest values in column "a" and then "c", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['a', 'c'])
    a  b    c
```

(continues on next page)

(continued from previous page)

```
3  10  c  3.0
1  10  b  2.0
2   8  d  NaN
```

Attempting to use `nlargest` on non-numeric dtypes will raise a `TypeError`:

```
>>> df.nlargest(3, 'b')
Traceback (most recent call last):
TypeError: Column 'b' has dtype object, cannot use method 'nlargest'
```

notna()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

notnull()
Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or `numpy.NaN`, get mapped to False values.

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

DataFrame.notnull : alias of notna DataFrame.isna : boolean inverse of notna DataFrame.dropna : omit axes labels with missing values notna : top-level notna

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

nsmallest(*n*, *columns*, *keep='first'*)
Get the rows of a DataFrame sorted by the *n* smallest values of *columns*.

n [int] Number of items to retrieve

columns [list or str] Column name or names to order by

keep [{'first', 'last'}, default 'first'] Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

DataFrame

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 11, -1],
...                    'b': list('abdce'),
...                    'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nsmallest(3, 'a')
   a  b   c
4 -1  e   4
0  1  a   1
2  8  d NaN
```

nunique (*axis=0*, *dropna=True*)

Return Series with number of distinct observations over requested axis.

New in version 0.20.0.

axis : {0 or 'index', 1 or 'columns'}, default 0 dropna : boolean, default True

Don't include NaN in the counts.

nunique : Series

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 1, 1]})
>>> df.nunique()
A    3
B    1
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
```

pct_change (*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

periods [int, default 1] Periods to shift for forming percent change.

fill_method [str, default 'pad'] How to handle NAs before computing percent changes.

limit [int, default None] The number of consecutive NAs to fill before stopping.

freq [DateOffset, timedelta, or offset alias string, optional] Increment to use from time series API (e.g. 'M' or BDay()).

****kwargs** Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

chg [Series or DataFrame] The same type as the calling object.

Series.diff : Compute the difference of two elements in a Series. DataFrame.diff : Compute the difference of two elements in a DataFrame. Series.shift : Shift the index by some number of periods. DataFrame.shift : Shift the index by some number of periods.

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
```

(continues on next page)

(continued from previous page)

```
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
                FR      GR      IT
1980-01-01  4.0405  1.7246  804.74
1980-02-01  4.0963  1.7482  810.01
1980-03-01  4.3149  1.8519  860.13
```

```
>>> df.pct_change()
                   FR        GR        IT
1980-01-01        NaN       NaN       NaN
1980-02-01   0.013810  0.013684  0.006549
1980-03-01   0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...      '2016': [1769950, 30586265],
...      '2015': [1500923, 40912316],
...      '2014': [1371819, 41403351]},
...      index=['GOOG', 'APPL'])
>>> df
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

pipe (*func*, **args*, ***kwargs*)

Apply func(self, *args, **kwargs)

func [function] function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a `(callable, data_keyword)` tuple where `data_keyword` is a string indicating the keyword of `callable` that expects the NDFrame.

args [iterable, optional] positional arguments passed into `func`.

kwargs [mapping, optional] a dictionary of keyword arguments passed into `func`.

object : the return type of `func`.

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe((f, 'arg2'), arg1=a, arg3=c)
...  )
```

pandas.DataFrame.apply pandas.DataFrame.applymap pandas.Series.map

pivot (*index=None*, *columns=None*, *values=None*)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the User Guide for more on reshaping.

index [string or object, optional] Column to use to make new frame's index. If None, uses existing index.

columns [string or object] Column to use to make new frame's columns.

values [string, object or a list of the previous, optional] Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Changed in version 0.23.0: Also accept list of column names.

DataFrame Returns reshaped DataFrame.

ValueError: When there are any *index*, *columns* combinations with multiple values. *DataFrame.pivot_table* when you need to aggregate.

DataFrame.pivot_table [generalization of pivot that can handle] duplicate values for one index/column pair.

DataFrame.unstack [pivot based on the index values instead of a] column.

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                            'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
4   two   B    5    w
5   two   C    6    t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar  A   B   C
foo
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz       zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                    "bar": ['A', 'A', 'B', 'C'],
```

(continues on next page)

(continued from previous page)

```
...                    "baz": [1, 2, 3, 4]})
>>> df
   foo bar  baz
0  one   A    1
1  one   A    2
2  two   B    3
3  two   C    4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
   ...
ValueError: Index contains duplicate entries, cannot reshape
```

pivot_table(*values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*, *dropna=True*, *margins_name='All'*)

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame

values : column to aggregate, optional index : column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

columns [column, Grouper, array, or list of the previous] If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

aggfunc [function, list of functions, dict, default numpy.mean] If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions

fill_value [scalar, default None] Value to replace missing values with

margins [boolean, default False] Add all row / columns (e.g. for subtotal / grand totals)

dropna [boolean, default True] Do not include columns whose entries are all NaN

margins_name [string, default 'All'] Name of the row / column that will contain the totals when margins is True.

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                          "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                          "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                          "small", "large", "small", "small",
...                          "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
>>> df
     A    B      C  D
0  foo  one  small  1
1  foo  one  large  2
2  foo  one  large  2
```

(continues on next page)

(continued from previous page)

```
3  foo  two  small  3
4  foo  two  small  3
5  bar  one  large  4
6  bar  one  small  5
7  bar  two  small  6
8  bar  two  large  7
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                     columns=['C'], aggfunc=np.sum)
>>> table
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                     aggfunc={'D': np.mean,
...                              'E': [min, max, np.mean]})
>>> table
                  D   E
               mean max median min
A   C
bar large  5.500000  16   14.5  13
    small  5.500000  15   14.5  14
foo large  2.000000  10    9.5   9
    small  2.333333  12   11.0   8
```

table : DataFrame

DataFrame.pivot [pivot without aggregation that can handle] non-numeric data

plot

alias of `pandas.plotting._core.FramePlotMethods`

pop(*item*)

Return item and drop from frame. Raise KeyError if not found.

item [str] Column label to be popped

popped : Series

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
```

(continues on next page)

(continued from previous page)

```
...                   columns=('name', 'class', 'max_speed'))
>>> df
     name   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

```
>>> df.pop('class')
0      bird
1      bird
2    mammal
3    mammal
Name: class, dtype: object
```

```
>>> df
     name  max_speed
0  falcon      389.0
1  parrot       24.0
2    lion       80.5
3  monkey        NaN
```

pow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rpow

prod(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)

Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

product (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the product of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

prod : Series or DataFrame (if level specified)

By default, the product of an empty or all-NA Series is `1`

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

quantile(*q=0.5*, *axis=0*, *numeric_only=True*, *interpolation='linear'*)

Return values at the given quantile over requested axis, a la numpy.percentile.

q [float or array-like, default 0.5 (50% quantile)] 0 <= q <= 1, the quantile(s) to compute

axis [{0, 1, 'index', 'columns'} (default 0)] 0 or 'index' for row-wise, 1 or 'columns' for column-wise

numeric_only [boolean, default True] If False, the quantile of datetime and timedelta data will be computed as well

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] New in version 0.18.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint: (*i* + *j*) / 2.

quantiles : Series or DataFrame

- If `q` is an array, a DataFrame will be returned where the index is `q`, the columns are the columns of self, and the values are the quantiles.
- If `q` is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                      columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
                       'B': [pd.Timestamp('2010'),
                             pd.Timestamp('2011')],
                       'C': [pd.Timedelta('1 days'),
                             pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
```

(continues on next page)

(continued from previous page)

```
C        1 days 12:00:00
Name: 0.5, dtype: object
```

pandas.core.window.Rolling.quantile

query(*expr*, *inplace=False*, ***kwargs*)

Query the columns of a frame with a boolean expression.

expr [string] The query string to evaluate. You can refer to variables in the environment by prefixing them with an '@' character like `@a + b`.

inplace [bool] Whether the query should modify the data in place or return a modified copy

New in version 0.18.0.

kwargs [dict] See the documentation for `pandas.eval()` for complete details on the keyword arguments accepted by `DataFrame.query()`.

q : DataFrame

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level `pandas.eval()` function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the `&` and `|` (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the `query` documentation in indexing.

pandas.eval DataFrame.eval

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = pd.DataFrame(randn(10, 2), columns=list('ab'))
>>> df.query('a > b')
>>> df[df.a > df.b]  # same result as the previous expression
```

radd(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

DataFrame.add

rank (*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)
Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values

axis [{0 or 'index', 1 or 'columns'}, default 0] index to direct ranking

method [{'average', 'min', 'max', 'first', 'dense'}]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

numeric_only [boolean, default None] Include only float, int, boolean data. Valid only for DataFrame or Panel objects

na_option [{'keep', 'top', 'bottom'}]

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

ascending [boolean, default True] False for ranks by high (1) to low (N)

pct [boolean, default False] Computes percentage rank of data

ranks : same type as caller

rdiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

reindex(***kwargs*)
Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like, optional] New labels / index to conform the axis specified by 'axis' to.

index, columns [array-like, optional (should be specified using keywords)] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis [int or str, optional] Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

method [{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional] method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any "compatible" value

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...       'http_status': [200,200,404,404,301],
...       'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...        index=index)
>>> df
            http_status  response_time
Firefox             200           0.04
Chrome              200           0.02
Safari              404           0.07
IE10                404           0.08
Konqueror           301           1.00
```

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
IE10                   404           0.08
Chrome                 200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
```

(continues on next page)

(continued from previous page)

```
IE10                  404          0.08
Chrome                200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            prices
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
            prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to backpropagate the last valid value to fill the NaN values, pass bfill as an argument to the method keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29     100
2009-12-30     100
2009-12-31     100
2010-01-01     100
2010-01-02     101
2010-01-03     NaN
2010-01-04     100
2010-01-05      89
2010-01-06      88
2010-01-07     NaN
```

Please note that the NaN value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the NaN values present in the original dataframe, use the fillna() method.

See the user guide for more.

reindexed : DataFrame

reindex_axis (*labels*, *axis=0*, *method=None*, *level=None*, *copy=True*, *limit=None*, *fill_value=nan*)

Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and copy=False

labels [array-like] New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis : {0 or 'index', 1 or 'columns'} method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional

Method to use for filling holes in reindexed DataFrame:

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

copy [boolean, default True] Return a new object, even if the passed indexes are the same

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

limit [int, default None] Maximum number of consecutive elements to forward or backward fill

tolerance [optional] Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation abs(index[indexer] - target) <= tolerance.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

reindex, reindex_like

reindexed : DataFrame

reindex_like(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)
Return an object with matching indices to myself.

other : Object method : string or None copy : boolean, default True limit : int, default None

Maximum number of consecutive labels to fill for inexact matches.

tolerance [optional] Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

reindexed : same as input

rename(***kwargs*)
Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the user guide for more.

mapper, index, columns [dict-like or function, optional] dict-like or functions transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

axis [int or str, optional] Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

copy [boolean, default True] Also copy underlying data

inplace [boolean, default False] Whether to return a new DataFrame. If True then value of copy is ignored.

level [int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

renamed : DataFrame

pandas.DataFrame.rename_axis

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
```

(continues on next page)

(continued from previous page)

```
1  2  5
2  3  6
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

rename_axis (*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter the name of the index or columns.

mapper [scalar, list-like, optional] Value to set as the axis name attribute.

axis [{0 or 'index', 1 or 'columns'}, default 0] The index or the name of the axis.

copy [boolean, default True] Also copy underlying data.

inplace [boolean, default False] Modifies the object directly, instead of creating a new Series or DataFrame.

renamed [Series, DataFrame, or None] The same type as the caller or None if *inplace* is True.

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

pandas.Series.rename : Alter Series index labels or name pandas.DataFrame.rename : Alter DataFrame index labels or name pandas.Index.rename : Set new names on index

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s.rename_axis("foo")
foo
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")
     A  B
foo
0    1  4
1    2  5
2    3  6
```

```
>>> df.rename_axis("bar", axis="columns")
bar  A  B
0    1  4
1    2  5
2    3  6
```

reorder_levels (*order*, *axis=0*)

Rearrange index levels using input order. May not drop or duplicate levels

order [list of int or list of str] List representing new level order. Reference level by number (position) or by key (label).

axis [int] Where to reorder levels.

type of caller (new object)

replace (*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)

Replace values given in *to_replace* with *value*.

Values of the DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

to_replace [str, regex, list, dict, Series, int, float, or None] How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to_replace* will be replaced with *value*
  - str: string exactly matching *to_replace* will be replaced with *value*
  - regex: regexs matching *to_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to_replace* and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{'a': {'b': np.nan}}`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be `None` to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:

- This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

value [scalar, dict, list, str, regex, default None] Value to replace any values matching *to_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

inplace [boolean, default False] If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

limit [int, default None] Maximum size gap to forward or backward fill.

regex [bool or same types as *to_replace*, default False] Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to_replace* must be `None`.

method [{'pad', 'ffill', 'bfill', *None*}] The method to use when for replacement, when *to_replace* is a scalar, list or tuple and *value* is `None`.

Changed in version 0.23.0: Added to DataFrame.

DataFrame.fillna : Fill NA values DataFrame.where : Replace values based on boolean condition Series.str.replace : Simple string replacement.

DataFrame Object after replacement.

AssertionError

- If *regex* is not a `bool` and *to_replace* is not `None`.

TypeError

- If *to_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to_replace* does not match the type of the value being replaced

ValueError

- If a `list` or an `ndarray` is passed to *to_replace* and *value* but they are not the same length.

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to_replace* value, it is like key(s) in the dict are the to_replace part and value(s) in the dict are the value parameter.

Scalar ‘to_replace‘ and ‘value‘

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like ‘to_replace‘

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
2    3
3    3
4    4
dtype: int64
```

dict-like ‘to_replace‘

```
>>> df.replace({0: 10, 1: 100})
     A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
     A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
     A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression 'to_replace'

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
      A    B
0   new  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
      A    B
0   new  abc
1   foo  new
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$':'new', 'foo':'xyz'})
      A    B
0   new  abc
1   xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
      A    B
0   new  abc
1   new  new
2  bait  xyz
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the *to_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
```

(continues on next page)

(continued from previous page)

```
>>> df.replace({'a string': 'new value', True: False})  # raises
Traceback (most recent call last):
    ...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the pecularities of the *to_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1    None
2    None
3       b
4    None
dtype: object
```

When `value=None` and *to_replace* is a scalar, list or tuple, *replace* uses the method parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0    10
1    10
2    10
3     b
4     b
dtype: object
```

resample(*rule*, *how=None*, *axis=0*, *fill_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the on or level keyword.

rule [string] the offset string or object representing target conversion

axis : int, optional, default 0 closed : {'right', 'left'}

Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

label [{'right', 'left'}] Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

convention [{'start', 'end', 's', 'e'}] For PeriodIndex only, controls whether to use the start or end of *rule*

kind: {'timestamp', 'period'}, optional Pass 'timestamp' to convert the resulting index to a `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`. By default the input representation is retained.

loffset [timedelta] Adjust the resampled time labels

base [int, default 0] For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

on [string, optional] For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

level [string or int, optional] For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

Resampler object

See the user guide for more.

To learn more about the offset strings, please see this link.

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5] #select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like)+5
```

```
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))
>>> s
2012    1
```

(continues on next page)

(continued from previous page)

```
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using 'start' *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
                     a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                       columns=['a', 'b', 'c', 'd'],
                       index=pd.MultiIndex.from_product([time, [1, 2]])
                       )
>>> df2.resample('3T', level=0).sum()
                     a  b   c   d
2000-01-01 00:00:00  0  6  12  18
2000-01-01 00:03:00  0  4   8  12
```

groupby : Group by mapping, function, label, or list of labels.

reset_index(*level=None*, *drop=False*, *inplace=False*, *col_level=0*, *col_fill=''*)

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level_0' (if 'index' is already taken) will be used.

level [int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default

drop [boolean, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

col_level [int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

col_fill [object, default ''] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

resetted : DataFrame

```
>>> df = pd.DataFrame([('bird',    389.0),
...                    ('bird',     24.0),
...                    ('mammal',   80.5),
...                    ('mammal', np.nan)],
...                   index=['falcon', 'parrot', 'lion', 'monkey'],
...                   columns=('class', 'max_speed'))
>>> df
         class  max_speed
falcon    bird      389.0
parrot    bird       24.0
lion    mammal       80.5
monkey  mammal        NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
    class  max_speed
0    bird      389.0
1    bird       24.0
2  mammal       80.5
3  mammal        NaN
```

You can also use *reset_index* with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                    ('bird', 'parrot'),
...                                    ('mammal', 'lion'),
...                                    ('mammal', 'monkey')],
```

(continues on next page)

(continued from previous page)

```
...                                       names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                      ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
...                    ( 24.0, 'fly'),
...                    ( 80.5, 'run'),
...                    (np.nan, 'jump')],
...                   index=index,
...                   columns=columns)
>>> df
               speed species
                 max    type
class  name
bird   falcon  389.0     fly
       parrot   24.0     fly
mammal lion     80.5     run
       monkey    NaN    jump
```

If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
         class  speed species
                  max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
                speed species
         class    max    type
name
falcon    bird  389.0     fly
parrot    bird   24.0     fly
lion    mammal   80.5     run
monkey  mammal    NaN    jump
```

When the index is inserted under another level, we can specify under which one with the parameter *col_fill*:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
              species  speed species
                class    max    type
name
falcon           bird  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

If we specify a nonexistent level for *col_fill*, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
                genus  speed species
                class    max    type
```

(continues on next page)

(continued from previous page)

```
name
falcon          bird  389.0     fly
parrot          bird   24.0     fly
lion          mammal   80.5     run
monkey        mammal    NaN    jump
```

rfloordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.floordiv

rmod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mod

rmul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.mul

rolling(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculations.

New in version 0.18.0.

window [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. This is new in 0.19.0

min_periods [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

center [boolean, default False] Set the labels at the center of the window.

win_type [string, default None] Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

on [string, optional] For a DataFrame, column on which to calculate the rolling window, rather than the index

closed [string, default None] Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

axis : int or string, default 0

a Window or Rolling sub-classed for the particular operation

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
     B
0  NaN
1  1.0
```

(continues on next page)

(continued from previous page)

```
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                   index = [pd.Timestamp('20130101 09:00:00'),
...                            pd.Timestamp('20130101 09:00:02'),
...                            pd.Timestamp('20130101 09:00:03'),
...                            pd.Timestamp('20130101 09:00:05'),
...                            pd.Timestamp('20130101 09:00:06')])
```

```
>>> df
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```
>>> df.rolling('2s').sum()
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

To learn more about the offsets & frequency strings, please see this link.

The recognized win_types are:

- boxcar
- triang
- blackman
- hamming
- bartlett
- parzen
- bohman
- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general_gaussian (needs power, width)
- slepian (needs width).

If win_type=None all points are evenly weighted. To learn more about different window types see scipy.signal window functions.

expanding : Provides expanding transformations. ewm : Provides exponential weighted functions

round(*decimals=0*, **args*, ***kwargs*)

Round a DataFrame to a variable number of decimal places.

decimals [int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

```
>>> df = pd.DataFrame(np.random.random([3, 3]),
...     columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
>>> df
                A         B         C
first    0.028208  0.992815  0.173891
second   0.038683  0.645646  0.577595
third    0.877076  0.149370  0.491027
>>> df.round(2)
           A     B     C
first   0.03  0.99  0.17
second  0.04  0.65  0.58
third   0.88  0.15  0.49
>>> df.round({'A': 1, 'C': 2})
          A         B     C
first   0.0  0.992815  0.17
second  0.0  0.645646  0.58
third   0.9  0.149370  0.49
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
          A  B     C
first   0.0  1  0.17
```

(continues on next page)

(continued from previous page)

```
second  0.0  1  0.58
third   0.9  0  0.49
```

DataFrame object

numpy.around Series.round

rpow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.pow

rsub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
```

(continues on next page)

(continued from previous page)

```
...                    index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.sub

rtruediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.truediv

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*)

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

n [int, optional] Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

frac [float, optional] Fraction of axis items to return. Cannot be used with *n*.

replace [boolean, optional] Sample with or without replacement. Default = False.

weights [str or ndarray-like, optional] Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. inf and -inf values not allowed.

random_state [int or numpy.random.RandomState, optional] Seed for the random number generator (if int), or numpy RandomState object.

axis [int or string, optional] Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

A new object of same type as caller.

Generate an example `Series` and `DataFrame`:

```
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0   -0.038497
1    1.820773
2   -0.972766
3   -1.598270
4   -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
          A         B         C         D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661
```

Next extract a random sample from both of these objects...

3 random elements from the `Series`:

```
>>> s.sample(n=3)
27   -0.994689
55   -1.049016
67   -0.224565
dtype: float64
```

And a random 10% of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.1, replace=True)
           A         B         C         D
35  1.981780  0.142106  1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640  0.217116
40  0.823173 -0.078816  1.009536  1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
6  -0.148339  0.832938  1.787600 -1.383767
```

You can use *random state* for reproducibility:

```
>>> df.sample(random_state=1)
A         B         C         D
37 -2.027662  0.103611  0.237496 -0.165867
43 -0.259323 -0.583426  1.516140 -0.479118
12 -1.686325 -0.579510  0.985195 -0.460286
8   1.167946  0.429082  1.215742 -1.636041
9   1.197475 -0.864188  1.554031 -1.505264
```

select (*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use df.loc[df.index.map(crit)] to select via labels

crit [function] To be called on each index (label). Should return True or False

axis : int

selection : type of caller

select_dtypes(*include=None*, *exclude=None*)

Return a subset of the DataFrame's columns based on the column dtypes.

include, exclude [scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

ValueError

- If both of `include` and `exclude` are empty
- If `include` and `exclude` have overlapping elements
- If any kind of string dtype is passed in.

subset [DataFrame] The subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.

- To select all *numeric* types, use `np.number` or `'number'`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the numpy dtype hierarchy
- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`
- To select timedeltas, use `np.timedelta64`, `'timedelta'` or `'timedelta64'`
- To select Pandas categorical dtypes, use `'category'`
- To select Pandas datetimetz dtypes, use `'datetimetz'` (new in 0.20.0) or `'datetime64[ns, tz]'`

```
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
        a      b  c
0       1   True  1.0
1       2  False  2.0
2       1   True  1.0
3       2  False  2.0
4       1   True  1.0
5       2  False  2.0
```

```
>>> df.select_dtypes(include='bool')
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

```
>>> df.select_dtypes(include=['float64'])
   c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int'])
       b    c
0   True  1.0
1  False  2.0
2   True  1.0
3  False  2.0
4   True  1.0
5  False  2.0
```

sem(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

sem : Series or DataFrame (if level specified)

set_axis(*labels*, *axis=0*, *inplace=None*)
Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

labels [list-like, Index] The values for the new index.

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

inplace [boolean, default None] Whether to return a new %(klass)s instance.

> **Warning:** `inplace=None` currently falls back to to True, but in a future version, will default to False. Use inplace=True explicitly rather than relying on the default.

renamed [%(klass)s or None] An object of same type as caller if inplace=False, None otherwise.

pandas.DataFrame.rename_axis : Alter the name of the index or columns.

Series

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

set_index (*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

keys : column label or list of column labels / arrays drop : boolean, default True

Delete columns to be used as the new index

append [boolean, default False] Whether to append columns to existing index

inplace [boolean, default False] Modify the DataFrame in place (do not create a new object)

verify_integrity [boolean, default False] Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale':[55, 40, 84, 31]})
   month  sale  year
0  1      55    2012
1  4      40    2014
2  7      84    2013
3  10     31    2014
```

Set the index to become the 'month' column:

```
>>> df.set_index('month')
       sale  year
month
1      55    2012
4      40    2014
7      84    2013
10     31    2014
```

Create a multi-index using columns 'year' and 'month':

```
>>> df.set_index(['year', 'month'])
            sale
year  month
2012  1     55
2014  4     40
2013  7     84
2014  10    31
```

Create a multi-index using a set of values and a column:

```
>>> df.set_index([[1, 2, 3, 4], 'year'])
         month  sale
   year
1  2012  1      55
2  2014  4      40
3  2013  7      84
4  2014  10     31
```

dataframe : DataFrame

set_value(*index*, *col*, *value*, *takeable=False*)

Put single value at passed column and index

Deprecated since version 0.21.0: Use .at[] or .iat[] accessors instead.

index : row label col : column label value : scalar value takeable : interpret the index/col as indexers, default False

frame [DataFrame] If label pair is contained, will be reference to calling DataFrame, otherwise a new object

shape
Return a tuple representing the dimensionality of the DataFrame.

ndarray.shape

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

shift (*periods=1*, *freq=None*, *axis=0*)
Shift index by desired number of periods with an optional time freq

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, optional] Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

axis : {0 or 'index', 1 or 'columns'}

If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data.

shifted : DataFrame

size
Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

ndarray.size

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

skew (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis Normalized by N-1

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

skew : Series or DataFrame (if level specified)

slice_shift (*periods=1*, *axis=0*)
Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

periods [int] Number of periods to move, can be positive or negative

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

shifted : same type as caller

sort_index (*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)
Sort object by labels (along an axis)

axis : index, columns to direct sorting level : int or level name or list of ints or list of level names

if not None, sort on values in specified index level(s)

ascending [boolean, default True] Sort ascending vs. descending

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end. Not implemented for MultiIndex.

sort_remaining [bool, default True] if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

sorted_obj : DataFrame

sort_values (*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)
Sort by the values along either axis

by [str or list of str] Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels

Changed in version 0.23.0: Allow specifying index or column level names.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis to be sorted

ascending [bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

inplace [bool, default False] if True, perform operation in-place

kind [{'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'] Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] *first* puts NaNs at the beginning, *last* puts NaNs at the end

sorted_obj : DataFrame

```
>>> df = pd.DataFrame({
...     'col1' : ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2' : [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
```

(continues on next page)

(continued from previous page)

```
... })
>>> df
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
3   NaN  8    4
4   D    7    2
5   C    4    3
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
    col1 col2 col3
0   A    2    0
1   A    1    1
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
    col1 col2 col3
1   A    1    1
0   A    2    0
2   B    9    9
5   C    4    3
4   D    7    2
3   NaN  8    4
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
    col1 col2 col3
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
3   NaN  8    4
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
    col1 col2 col3
3   NaN  8    4
4   D    7    2
5   C    4    3
2   B    9    9
0   A    2    0
1   A    1    1
```

sortlevel (*level=0*, *axis=0*, *ascending=True*, *inplace=False*, *sort_remaining=True*)

Sort multilevel index by chosen axis and primary level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order).

Deprecated since version 0.20.0: Use `DataFrame.sort_index()`

level : int axis : {0 or 'index', 1 or 'columns'}, default 0 ascending : boolean, default True inplace : boolean, default False

Sort the DataFrame without creating a new instance

sort_remaining [boolean, default True] Sort by the other levels too.

sorted : DataFrame

DataFrame.sort_index(level=...)

squeeze (*axis=None*)

Squeeze length 1 dimensions.

axis [None, integer or string axis name, optional] The axis to squeeze if 1-sized.

New in version 0.20.0.

scalar if 1-sized, else original object

stack (*level=-1*, *dropna=True*)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

The new index levels are sorted.

level [int, str, list, default -1] Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

dropna [bool, default True] Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of index and column values that are missing from the original dataframe. See Examples section.

DataFrame or Series Stacked dataframe or series.

DataFrame.unstack [Unstack prescribed level(s) from index axis] onto column axis.

DataFrame.pivot [Reshape dataframe from long format to wide] format.

DataFrame.pivot_table [Create a spreadsheet-style pivot table] as a DataFrame.

The function is named by analogy with a collection of books being re-organised from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of of each other (in the index of the dataframe).

Single level columns

```
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
     weight height
cat       0      1
dog       2      3
>>> df_single_level_cols.stack()
cat  weight    0
     height    1
dog  weight    2
     height    3
dtype: int64
```

Multi level columns: simple case

```
>>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol1)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
     weight
         kg    pounds
cat       1         2
dog       2         4
>>> df_multi_level_cols1.stack()
            weight
cat kg           1
    pounds       2
dog kg           2
    pounds       4
```

Missing values

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                        ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
    weight height
        kg      m
cat    1.0    2.0
dog    3.0    4.0
>>> df_multi_level_cols2.stack()
        height  weight
cat kg     NaN     1.0
    m      2.0     NaN
dog kg     NaN     3.0
    m      4.0     NaN
```

Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0)
             kg    m
cat height  NaN  2.0
    weight  1.0  NaN
dog height  NaN  4.0
    weight  3.0  NaN
>>> df_multi_level_cols2.stack([0, 1])
cat  height  m     2.0
     weight  kg    1.0
dog  height  m     4.0
     weight  kg    3.0
dtype: float64
```

Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
    weight height
        kg      m
cat    NaN    1.0
dog    2.0    3.0
>>> df_multi_level_cols3.stack(dropna=False)
        height  weight
cat kg     NaN     NaN
    m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
>>> df_multi_level_cols3.stack(dropna=True)
        height  weight
cat m      1.0     NaN
dog kg     NaN     2.0
    m      3.0     NaN
```

std(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

std : Series or DataFrame (if level specified)

style
Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

pandas.io.formats.style.Styler

sub (*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

subtract (*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
c  1.0  NaN
d  -1.0  NaN
e  NaN  -2.0
```

DataFrame.rsub

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, *min_count=0*, ***kwargs*)
Return the sum of the values for the requested axis

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values when computing the result.

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

min_count [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

sum : Series or DataFrame (if level specified)

By default, the sum of an empty or all-NA Series is `0`.

```
>>> pd.Series([]).sum()  # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

swapaxes (*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately

y : same as input

swaplevel (*i=-2*, *j=-1*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

i, j [int, string (can be mixed)] Level of index to be swapped. Can pass level name as string.

swapped : type of caller (new object)

Changed in version 0.18.1: The indexes `i` and `j` are now optional, and default to the two innermost levels of the index.

tail (*n=5*)

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

n [int, default 5] Number of rows to select.

type of caller The last *n* rows of the caller object.

pandas.DataFrame.head : The first *n* rows of the caller object.

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

take (*indices*, *axis=0*, *convert=None*, *is_copy=True*, ***kwargs*)

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

indices [array-like] An array of ints indicating which positions to take.

axis [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. `0` means that we are selecting rows, `1` means that we are selecting columns.

convert [bool, default True] Whether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

is_copy [bool, default True] Whether to return a copy of the original object or not.

****kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

taken [type of caller] An array-like containing the elements taken from the object.

DataFrame.loc : Select a subset of a DataFrame by labels. DataFrame.iloc : Select a subset of a DataFrame by positions. numpy.take : Take elements from an array along an axis.

```
>>> df = pd.DataFrame([('falcon', 'bird',    389.0),
...                    ('parrot', 'bird',     24.0),
...                    ('lion',   'mammal',   80.5),
...                    ('monkey', 'mammal', np.nan)],
...                    columns=['name', 'class', 'max_speed'],
...                    index=[0, 2, 3, 1])
>>> df
     name   class  max_speed
0  falcon    bird      389.0
2  parrot    bird       24.0
3    lion  mammal       80.5
1  monkey  mammal        NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
     name   class  max_speed
0  falcon    bird      389.0
1  monkey  mammal        NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
    class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

to_clipboard(*excel=True*, *sep=None*, ***kwargs*)

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

excel [bool, default True]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

sep [str, default `'\t'`] Field delimiter.

****kwargs** These parameters will be passed to DataFrame.to_csv.

DataFrame.to_csv [Write a DataFrame to a comma-separated values] (csv) file.

read_clipboard : Read text from clipboard and pass to read_table.

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

to_csv(*path_or_buf=None*, *sep=', '*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *encoding=None*, *compression=None*, *quoting=None*, *quotechar='"'*, *line_terminator='\n'*, *chunksize=None*, *tupleize_cols=None*, *date_format=None*, *doublequote=True*, *escapechar=None*, *decimal='.'*)

Write DataFrame to a comma-separated values (csv) file

path_or_buf [string or file handle, default None] File path or object, if None is provided the result is returned as a string.

sep [character, default ','] Field delimiter for the output file.

na_rep [string, default ''] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, or False, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R

mode [str] Python write mode, default 'w'

encoding [string, optional] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

compression [string, optional] A string representing the compression to use in the output file. Allowed values are 'gzip', 'bz2', 'zip', 'xz'. This input is only used when the first argument is a filename.

line_terminator [string, default `'\n'`] The newline character or character sequence to use in the output file

quoting [optional constant from csv module] defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric

quotechar [string (length 1), default '"'] character used to quote fields

doublequote [boolean, default True] Control quoting of *quotechar* inside a field

escapechar [string (length 1), default None] character used to escape *sep* and *quotechar* when appropriate

chunksize [int or None] rows to write at a time

tupleize_cols [boolean, default False] Deprecated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

Write MultiIndex columns as a list of tuples (if True) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if False).

date_format [string, default None] Format string for datetime objects

decimal: string, default '.' Character recognized as decimal separator. E.g. use ',' for European data

to_dense()
Return dense representation of NDFrame (as opposed to sparse)

to_dict(*orient='dict'*, *into=<type 'dict'>*)
Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

orient [str {'dict', 'list', 'series', 'split', 'records', 'index'}] Determines the type of the values of the dictionary.

- 'dict' (default) : dict like {column -> {index -> value}}
- 'list' : dict like {column -> [values]}
- 'series' : dict like {column -> Series(values)}
- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'records' : list like [{column -> value}, ... , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

into [class, default dict] The collections.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

New in version 0.21.0.

result : collections.Mapping like {column -> {index -> value}}

DataFrame.from_dict: create a DataFrame from a dictionary DataFrame.to_json: convert a DataFrame to JSON format

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   col1  col2
a     1   0.50
b     2   0.75
>>> df.to_dict()
{'col1': {'a': 1, 'b': 2}, 'col2': {'a': 0.5, 'b': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': a    1
         b    2
         Name: col1, dtype: int64,
 'col2': a    0.50
         b    0.75
         Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['a', 'b'], 'columns': ['col1', 'col2'],
 'data': [[1.0, 0.5], [2.0, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1.0, 'col2': 0.5}, {'col1': 2.0, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'a': {'col1': 1.0, 'col2': 0.5}, 'b': {'col1': 2.0, 'col2': 0.75}}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('a', 1), ('b', 2)])),
             ('col2', OrderedDict([('a', 0.5), ('b', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1.0, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2.0, 'col2': 0.75})]
```

to_excel(*excel_writer*, *sheet_name='Sheet1'*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *startrow=0*, *startcol=0*, *engine=None*, *merge_cells=True*, *encoding=None*, *inf_rep='inf'*, *verbose=True*, *freeze_panes=None*)

Write DataFrame to an excel sheet

excel_writer [string or ExcelWriter object] File path or existing ExcelWriter

sheet_name [string, default 'Sheet1'] Name of sheet which will contain DataFrame

na_rep [string, default ''] Missing data representation

float_format [string, default None] Format string for floating point numbers

columns [sequence, optional] Columns to write

header [boolean or list of string, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

index [boolean, default True] Write row names (index)

index_label [string or sequence, default None] Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

startrow : upper left cell row to dump data frame

startcol : upper left cell column to dump data frame

engine [string, default None] write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

merge_cells [boolean, default True] Write MultiIndex and Hierarchical Rows as merged cells.

encoding: string, default None encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

inf_rep [string, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel)

freeze_panes [tuple of integer (length 2), default None] Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer,'Sheet1')
>>> df2.to_excel(writer,'Sheet2')
>>> writer.save()
```

For compatibility with to_csv, to_excel serializes lists and dicts to strings before writing.

to_feather(*fname*)
write out the binary feather-format for DataFrames

New in version 0.20.0.

fname [str] string file path

to_gbq(*destination_table*, *project_id*, *chunksize=None*, *verbose=None*, *reauth=False*, *if_exists='fail'*, *private_key=None*, *auth_local_webserver=False*, *table_schema=None*)
Write a DataFrame to a Google BigQuery table.

This function requires the pandas-gbq package.

Authentication to the Google BigQuery service is via OAuth 2.0.

- If `private_key` is provided, the library loads the JSON service account credentials and uses those to authenticate.
- If no `private_key` is provided, the library tries application default credentials.
- If application default credentials are not found or cannot be used with BigQuery, the library authenticates with user account credentials. In this case, you will be asked to grant permissions for product name 'pandas GBQ'.

destination_table [str] Name of table to be written, in the form 'dataset.tablename'.

project_id [str] Google BigQuery Account project ID.

chunksize [int, optional] Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

reauth [bool, default False] Force Google BigQuery to reauthenticate the user. This is useful if multiple accounts are used.

if_exists [str, default 'fail'] Behavior when the destination table exists. Value can be one of:

'fail' If table exists, do nothing.

'replace' If table exists, drop it, recreate it, and insert data.

'append' If table exists, insert data. Create if does not exist.

private_key [str, optional] Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

auth_local_webserver [bool, default False] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

table_schema [list of dicts, optional] List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'},...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

New in version 0.3.1 of pandas-gbq.

verbose [boolean, deprecated] *Deprecated in Pandas-GBQ 0.4.0.* Use the logging module to adjust verbosity instead.

pandas_gbq.to_gbq : This function in the pandas-gbq library. pandas.read_gbq : Read a DataFrame from Google BigQuery.

to_hdf (*path_or_buf*, *key*, ***kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the user guide.

path_or_buf [str or pandas.HDFStore] File path or HDFStore object.

key [str] Identifier for the group in the store.

mode [{'a', 'w', 'r+'}, default 'a'] Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

format [{'fixed', 'table'}, default 'fixed'] Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

append [bool, default False] For Table formats, append the input data to the existing.

data_columns [list of columns or True, optional] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See io.hdf5-query-data-columns. Applicable only to format='table'.

complevel [{0-9}, optional] Specifies a compression level for data. A value of 0 disables compression.

complib [{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'] Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

fletcher32 [bool, default False] If applying compression use the fletcher32 checksum.

dropna [bool, default False] If true, ALL nan rows will not be written to store.

errors [str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

DataFrame.read_hdf : Read from HDF file. DataFrame.to_parquet : Write a DataFrame to the binary parquet format. DataFrame.to_sql : Write to a sql table. DataFrame.to_feather : Write out feather-format for DataFrames. DataFrame.to_csv : Write out to a csv file.

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

to_html (*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *bold_rows=True*, *classes=None*, *escape=True*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*, *notebook=False*, *decimal='.'*, *border=None*, *table_id=None*)

Render a DataFrame as an HTML table.

to_html-specific options:

bold_rows [boolean, default True] Make the row labels bold in the output

classes [str or list or tuple, default None] CSS class(es) to apply to the resulting html table

escape [boolean, default True] Convert the characters <, >, and & to HTML-safe sequences.

max_rows [int, optional] Maximum number of rows to show before truncating. If None, show all.

max_cols [int, optional] Maximum number of columns to show before truncating. If None, show all.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe

New in version 0.18.0.

border [int] A `border=border` attribute is included in the opening <*table*> tag. Default `pd.options.html.border`.

New in version 0.19.0.

table_id [str, optional] A css id is included in the opening <*table*> tag if specified.

New in version 0.23.0.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] whether to print column labels, default True

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_json(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression=None*, *index=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

path_or_buf [string or file handle, optional] File path or object. If not specified, the result is returned as a string.

orient [string] Indication of expected JSON string format.

- Series
  - default is 'index'

  - allowed values are: {‘split’,’records’,’index’}
- DataFrame
  - default is ‘columns’
  - allowed values are: {‘split’,’records’,’index’,’columns’,’values’}
- The format of the JSON string
  - ‘split’ : dict like {‘index’ -> [index], ‘columns’ -> [columns], ‘data’ -> [values]}
  - ‘records’ : list like [{column -> value}, … , {column -> value}]
  - ‘index’ : dict like {index -> {column -> value}}
  - ‘columns’ : dict like {column -> {index -> value}}
  - ‘values’ : just the values array
  - ‘table’ : dict like {‘schema’: {schema}, ‘data’: {data}} describing the data, and the data component is like `orient='records'`.

    Changed in version 0.20.0.

date_format [{None, ‘epoch’, ‘iso’}] Type of date conversion. ‘epoch’ = epoch milliseconds, ‘iso’ = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is ‘iso’. For all other orients, the default is ‘epoch’.

double_precision [int, default 10] The number of decimal places to use when encoding floating point values.

force_ascii [boolean, default True] Force encoded string to be ASCII.

date_unit [string, default ‘ms’ (milliseconds)] The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’, ‘ns’ for second, millisecond, microsecond, and nanosecond respectively.

default_handler [callable, default None] Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

lines [boolean, default False] If ‘orient’ is ‘records’ write out line delimited json format. Will throw ValueError if incorrect ‘orient’ since others are not list like.

New in version 0.19.0.

compression [{None, ‘gzip’, ‘bz2’, ‘zip’, ‘xz’}] A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

index [boolean, default True] Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is ‘split’ or ‘table’.

New in version 0.23.0.

pandas.read_json

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]}'
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
             "primaryKey": "index",
             "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
           {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

to_latex(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=False*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires \usepackage{booktabs}.

Changed in version 0.20.2: Added to Series

to_latex-specific options:

bold_rows [boolean, default False] Make the row labels bold in the output

column_format [str, default None] The columns format as specified in LaTeX table format e.g 'rcl' for 3 columns

longtable [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a \usepackage{longtable} to your LaTeX preamble.

escape [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

encoding [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

decimal [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

multicolumn [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

multicolumn_format [str, default 'l'] The alignment for multicolumns, similar to *column_format* The default will be read from the config module.

New in version 0.20.0.

multirow [boolean, default False] Use multirow to enhance MultiIndex rows. Requires adding a \usepackage{multirow} to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

New in version 0.20.0.

to_msgpack (*path_or_buf=None*, *encoding='utf-8'*, ***kwargs*)
msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

path [string File path, buffer-like, or None] if None, return generated string

append [boolean whether to append to an existing msgpack] (default is False)

compress [type of compressor (zlib or blosc), default to None (no] compression)

to_panel ()
Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Deprecated since version 0.20.0.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

panel : Panel

to_parquet (*fname*, *engine='auto'*, *compression='snappy'*, ***kwargs*)
Write a DataFrame to the binary parquet format.

New in version 0.21.0.

This function writes the dataframe as a parquet file. You can choose different parquet backends, and have the option of compression. See the user guide for more details.

fname [str] String file path.

engine [{'auto', 'pyarrow', 'fastparquet'}, default 'auto'] Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

compression [{'snappy', 'gzip', 'brotli', None}, default 'snappy'] Name of the compression to use. Use `None` for no compression.

****kwargs** Additional arguments passed to the parquet library. See pandas io for more details.

read_parquet : Read a parquet file. DataFrame.to_csv : Write a csv file. DataFrame.to_sql : Write to a sql table. DataFrame.to_hdf : Write to hdf.

This function requires either the fastparquet or pyarrow library.

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gzip', compression='gzip')
>>> pd.read_parquet('df.parquet.gzip')
   col1  col2
```

(continues on next page)

(continued from previous page)

```
0      1      3
1      2      4
```

to_period(*freq=None*, *axis=0*, *copy=True*)

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

freq : string, default axis : {0 or 'index', 1 or 'columns'}, default 0

The axis to convert (the index by default)

copy [boolean, default True] If False then underlying input data is not copied

ts : TimeSeries with PeriodIndex

to_pickle(*path*, *compression='infer'*, *protocol=2*)

Pickle (serialize) object to file.

path [str] File path where the pickled object will be stored.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

protocol [int] Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1]_ paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python>=3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

New in version 0.21.0.

read_pickle : Load pickled pandas object (or any object) from file. DataFrame.to_hdf : Write DataFrame to an HDF5 file. DataFrame.to_sql : Write DataFrame to a SQL database. DataFrame.to_parquet : Write a DataFrame to the binary parquet format.

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

to_records(*index=True*, *convert_datetime64=None*)

Convert DataFrame to a NumPy record array.

Index will be put in the 'index' field of the record array if requested.

index [boolean, default True] Include index in resulting record array, stored in 'index' field.

convert_datetime64 [boolean, default None] Deprecated since version 0.23.0.

Whether to convert the index to datetime.datetime if it is a DatetimeIndex.

y : numpy.recarray

DataFrame.from_records: convert structured or record ndarray to DataFrame.

numpy.recarray: ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                   index=['a', 'b'])
>>> df
   A     B
a  1  0.50
b  2  0.75
>>> df.to_records()
rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

By default, timestamps are converted to *datetime.datetime*:

```
>>> df.index = pd.date_range('2018-01-01 09:00', periods=2, freq='min')
>>> df
                     A     B
2018-01-01 09:00:00  1  0.50
2018-01-01 09:01:00  2  0.75
>>> df.to_records()
rec.array([(datetime.datetime(2018, 1, 1, 9, 0), 1, 0.5 ),
           (datetime.datetime(2018, 1, 1, 9, 1), 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The timestamp conversion can be disabled so NumPy's datetime64 data type is used instead:

```
>>> df.to_records(convert_datetime64=False)
rec.array([('2018-01-01T09:00:00.000000000', 1, 0.5 ),
           ('2018-01-01T09:01:00.000000000', 2, 0.75)],
          dtype=[('index', '<M8[ns]'), ('A', '<i8'), ('B', '<f8')])
```

to_sparse(*fill_value=None*, *kind='block'*)

Convert to SparseDataFrame

fill_value : float, default NaN kind : {'block', 'integer'}

y : SparseDataFrame

to_sql(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1]_ are supported. Tables can be newly created, appended to, or overwritten.

name [string] Name of SQL table.

con [sqlalchemy.engine.Engine or sqlite3.Connection] Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

schema [string, optional] Specify the schema (if database flavor supports this). If None, use default schema.

if_exists [{'fail', 'replace', 'append'}, default 'fail'] How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index [boolean, default True] Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label [string or sequence, default None] Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize [int, optional] Rows will be written in batches of this size at a time. By default, all rows will be written at once.

dtype [dict, optional] Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

ValueError When the table already exists and *if_exists* is 'fail' (the default).

pandas.read_sql : read a DataFrame from a table

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
```

(continues on next page)

(continued from previous page)

```
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

to_stata(*fname*, *convert_dates=None*, *write_index=True*, *encoding='latin-1'*, *byteorder=None*, *time_stamp=None*, *data_label=None*, *variable_labels=None*, *version=114*, *convert_strl=None*)

Export Stata binary dta files.

fname [path (string), buffer or path object] string, path object (pathlib.Path or py._path.local.LocalPath) or object implementing a binary write() functions. If using a buffer then the buffer will not be automatically closed after the file data has been written.

convert_dates [dict] Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to 'tc'. Raises NotImplementedError if a datetime column has timezone information.

write_index [bool] Write the index to Stata dataset.

encoding [str] Default is latin-1. Unicode is not supported.

byteorder [str] Can be ">", "<", "little", or "big". default is *sys.byteorder*.

time_stamp [datetime] A datetime to use as file creation date. Default is the current time.

data_label [str] A label for the data set. Must be 80 characters or smaller.

variable_labels [dict] Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

New in version 0.19.0.

version [{114, 117}] Version to use in the output dta file. Version 114 can be used read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 114 limits string variables to 244 characters

or fewer while 117 allows strings with lengths up to 2,000,000 characters.

New in version 0.23.0.

convert_strl [list, optional] List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

New in version 0.23.0.

NotImplementedError

- If datetimes contain timezone information
- Column dtype is not representable in Stata

ValueError

- Columns listed in convert_dates are neither datetime64[ns] or datetime.datetime
- Column listed in convert_dates is not in DataFrame
- Categorical label contains more than 32,000 characters

New in version 0.19.0.

pandas.read_stata : Import Stata data files pandas.io.stata.StataWriter : low-level writer for Stata data files pandas.io.stata.StataWriter117 : low-level writer for version 117 files

```
>>> data.to_stata('./data_file.dta')
```

Or with dates

```
>>> data.to_stata('./date_data_file.dta', {2 : 'tw'})
```

Alternatively you can create an instance of the StataWriter class

```
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

With dates:

```
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
```

to_string(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *line_width=None*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*)

Render a DataFrame to a console-friendly tabular output.

buf [StringIO-like, optional] buffer to write to

columns [sequence, optional] the subset of columns to write; default None writes all columns

col_space [int, optional] the minimum width of each column

header [bool, optional] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names

index [bool, optional] whether to print index (row) labels, default True

na_rep [string, optional] string representation of NAN to use, default 'NaN'

formatters [list or dict of one-parameter functions, optional] formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

float_format [one-parameter function, optional] formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

sparsify [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

index_names [bool, optional] Prints the names of the indexes, default True

line_width [int, optional] Width to wrap a line in characters, default no wrap

table_id [str, optional] id for the <table> element create by to_html

New in version 0.23.0.

justify [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

formatted : string (or unicode, depending on data and options)

to_timestamp (*freq=None*, *how='start'*, *axis=0*, *copy=True*)
Cast to DatetimeIndex of timestamps, at *beginning* of period

freq [string, default frequency of PeriodIndex] Desired frequency

how [{'s', 'e', 'start', 'end'}] Convention for converting period to timestamp; start of period vs. end

axis [{0 or 'index', 1 or 'columns'}, default 0] The axis to convert (the index by default)

copy [boolean, default True] If false then underlying input data is not copied

df : DataFrame with DatetimeIndex

to_xarray()
Return an xarray object from the pandas object.

a DataArray for a Series a Dataset for a DataFrame a DataArray for higher dims

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
```

(continues on next page)

(continued from previous page)

```
   A    B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index    (index) int64 0 1 2
Data variables:
    A        (index) int64 1 1 2
    B        (index) object 'foo' 'bar' 'foo'
    C        (index) float64 4.0 5.0 6.0
```

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)}
                      ).set_index(['B','A'])
>>> df
         C
B   A
foo 1  4.0
bar 1  5.0
foo 2  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B        (B) object 'bar' 'foo'
  * A        (A) int64 1 2
Data variables:
    C        (B, A) float64 5.0 nan 4.0 6.0
```

```
>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                 items=list('ABCD'),
                 major_axis=pd.date_range('20130101', periods=3),
                 minor_axis=['first', 'second'])
>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second
```

```
>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],
       [[ 6,  7],
        [ 8,  9],
        [10, 11]],
       [[12, 13],
```

(continues on next page)

(continued from previous page)

```
        [14, 15],
        [16, 17]],
       [[18, 19],
        [20, 21],
        [22, 23]]])
Coordinates:
  * items       (items) object 'A' 'B' 'C' 'D'
  * major_axis  (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03
↪# noqa
  * minor_axis  (minor_axis) object 'first' 'second'
```

See the xarray docs

transform(*func*, **args*, ***kwargs*)

Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values

New in version 0.20.0.

func [callable, string, dictionary, or list of string/callables] To apply to column

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

transformed : NDFrame

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
...                   index=pd.date_range('1/1/2000', periods=10))
df.iloc[3:7] = np.nan
```

```
>>> df.transform(lambda x: (x - x.mean()) / x.std())
                   A         B         C
2000-01-01  0.579457  1.236184  0.123424
2000-01-02  0.370357 -0.605875 -1.231325
2000-01-03  1.455756 -0.277446  0.288967
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -0.498658  1.274522  1.642524
2000-01-09 -0.540524 -1.012676 -0.828968
2000-01-10 -1.366388 -0.614710  0.005378
```

pandas.NDFrame.aggregate pandas.NDFrame.apply

transpose(**args*, ***kwargs*)

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

copy [bool, default False] If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

***args, **kwargs** Additional keywords have no effect but might be accepted for compatibility with numpy.

DataFrame The transposed DataFrame.

numpy.transpose : Permute the dimensions of a given array.

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
    name  score  employed  kids
0  Alice    9.5     False     0
1    Bob    8.0      True     0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
                0     1
name        Alice   Bob
score         9.5     8
employed    False  True
kids            0     0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name         object
score       float64
employed       bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

truediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

other : Series, DataFrame, or constant axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level [int or name] Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value [None or float value, default None] Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Mismatched indices will be unioned together

result : DataFrame

None

DataFrame.rtruediv

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

before [date, string, int] Truncate all rows before this index value.

after [date, string, int] Truncate all rows after this index value.

axis [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

copy [boolean, default is True,] Return a copy of the truncated section.

type of caller The truncated Series or DataFrame.

DataFrame.loc : Select a subset of a DataFrame by label. DataFrame.iloc : Select a subset of a DataFrame by position.

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
```

(continues on next page)

(continued from previous page)

```
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

tshift (*periods=1*, *freq=None*, *axis=0*)
Shift the time index, using the index's frequency if available.

periods [int] Number of periods to move, can be positive or negative

freq [DateOffset, timedelta, or time rule string, default None] Increment to use from the tseries module or time rule (e.g. 'EOM')

axis [int or basestring] Corresponds to the axis that contains the Index

If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

shifted : NDFrame

tz_convert (*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

tz : string or pytz.timezone object axis : the axis to convert level : int, str, default None

If axis ia a MultiIndex, convert a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

TypeError If the axis is tz-naive.

tz_localize (*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*)
Localize tz-naive TimeSeries to target time zone.

tz : string or pytz.timezone object axis : the axis to localize level : int, str, default None

If axis ia a MultiIndex, localize a specific level. Otherwise must be None

copy [boolean, default True] Also make a copy of the underlying data

ambiguous ['infer', bool-ndarray, 'NaT', default 'raise']

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times

- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

TypeError If the TimeSeries is tz-aware and tz is not None.

unstack (*level=-1*, *fill_value=None*)

Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels. If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex). The level involved will automatically get sorted.

level [int, string, or list of these, default -1 (last level)] Level(s) of index to unstack, can pass level name

fill_value [replace NaN with this value if the unstack produces] missing values

New in version 0.18.0.

DataFrame.pivot : Pivot a table based on column values. DataFrame.stack : Pivot a level of the column labels (inverse operation

from *unstack*).

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                    ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a   1.0
     b   2.0
two  a   3.0
     b   4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a   b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
   one  two
a  1.0   3.0
b  2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a  1.0
     b  2.0
two  a  3.0
     b  4.0
dtype: float64
```

unstacked : DataFrame or Series

update (*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

other [DataFrame, or object coercible into a DataFrame] Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

join [{'left'}, default 'left'] Only left join is implemented, keeping the index and columns of the original object.

overwrite [bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.
- False: only update values that are NA in the original DataFrame.

filter_func [callable(1d-array) -> boolean 1d-array, optional] Can choose to replace values other than NA. Return True for values that should be updated.

raise_conflict [bool, default False] If True, will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

ValueError When *raise_conflict* is True and there's overlapping non-NA data.

dict.update : Similar method for dictionaries. DataFrame.merge : For column(s)-on-columns(s) operations.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
1  b  e
2  c  f
```

For Series, it's name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
```

(continues on next page)

(continued from previous page)

```
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A      B
0  1    4.0
1  2  500.0
2  3    6.0
```

values

Return a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

numpy.ndarray The values of the DataFrame.

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age':    [ 3,  29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3      94      31
1   29     170     115
>>> df.dtypes
age       int64
height    int64
weight    int64
dtype: object
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('monkey', np.nan, None)],
...                    columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name          object
max_speed    float64
rank          object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
```

(continues on next page)

(continued from previous page)

```
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

pandas.DataFrame.index : Retrievie the index labels pandas.DataFrame.columns : Retrieving the column names

var(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

axis : {index (0), columns (1)} skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level [int or level name, default None] If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

ddof [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

numeric_only [boolean, default None] Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

var : Series or DataFrame (if level specified)

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *errors='raise'*, *try_cast=False*, *raise_on_error=None*)
Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

cond [boolean NDFrame, array-like, or callable] Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

other [scalar, NDFrame, or callable] Entries where *cond* is False are replaced with corresponding value from *other*. If other is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

inplace [boolean, default False] Whether to perform the operation in place on the data

axis : alignment axis if needed, default None level : alignment level if needed, default None errors : str, {'raise', 'ignore'}, default 'raise'

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

try_cast [boolean, default False] try to cast the result back to the input type (if possible),

raise_on_error [boolean, default True] Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

wh : same type as caller

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in indexing.

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2    2.0
3    3.0
4    4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
```

(continues on next page)

(continued from previous page)

```
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A     B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

DataFrame.mask()

xs(*key*, *axis=0*, *level=None*, *drop_level=True*)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on the rows (axis=0).

key [object] Some label contained in the index, or partially in a MultiIndex

axis [int, default 0] Axis to retrieve cross-section on

level [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

drop_level [boolean, default True] If False, returns object with same levels as self.

```
>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
A    4
B    5
C    2
Name: a
>>> df.xs('C', axis=1)
a    2
b    9
c    3
Name: C
```

```
>>> df
                     A  B  C  D
first second third
bar   one    1       4  1  8  9
      two    1       7  5  5  0
baz   one    1       6  6  8  0
      three  2       5  3  5  3
>>> df.xs(('baz', 'three'))
       A  B  C  D
third
2      5  3  5  3
>>> df.xs('one', level=1)
             A  B  C  D
first third
bar   1      4  1  8  9
baz   1      6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
        A  B  C  D
```

(continues on next page)

(continued from previous page)

```
second
three   5  3  5  3
```

xs : Series or DataFrame

xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of xs functionality, see MultiIndex Slicers

1.1.7 Core.Transcript

class Fred2.Core.Transcript.**Transcript**(*seq*, *gene_id='unknown'*, *transcript_id=None*, *vars=None*)

Bases: *Fred2.Core.Base.MetadataLogger*, Bio.Seq.Seq

A Transcript is the mRNA sequence containing no or several *Fred2.Core.Variant.Variant*.

Note: For accessing and manipulating the sequence see also Bio.Seq.Seq (from Biopython)

Parameters

- **gene_id** (*str*) – Genome ID
- **transcript_id** (*str*) – *Transcript* RefSeqID
- **vars** (dict(int,:class:*Fred2.Core.Variant.Variant*)) – Dict of *Fred2.Core.Variant.Variant* for specific positions in the *Transcript*. key=position, value=Variant

back_transcribe()

Return the DNA sequence from an RNA sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG",
...                     IUPAC.unambiguous_rna)
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
>>> messenger_rna.back_transcribe()
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
```

Trying to back-transcribe a protein or DNA sequence raises an exception:

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.back_transcribe()
Traceback (most recent call last):
   ...
ValueError: Proteins cannot be back transcribed!
```

complement()

Return the complement sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_dna = Seq("CCCCCGATAG", IUPAC.unambiguous_dna)
```

(continues on next page)

(continued from previous page)

```
>>> my_dna
Seq('CCCCCGATAG', IUPACUnambiguousDNA())
>>> my_dna.complement()
Seq('GGGGGCTATC', IUPACUnambiguousDNA())
```

You can of course used mixed case sequences,

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("CCCCCgatA-GD", generic_dna)
>>> my_dna
Seq('CCCCCgatA-GD', DNAAlphabet())
>>> my_dna.complement()
Seq('GGGGGctaT-CH', DNAAlphabet())
```

Note in the above example, ambiguous character D denotes G, A or T so its complement is H (for C, T or A).

Trying to complement a protein sequence raises an exception.

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.complement()
Traceback (most recent call last):
   ...
ValueError: Proteins do not have complements!
```

count (*sub*, *start=0*, *end=9223372036854775807*)

Return a non-overlapping count, like that of a python string.

This behaves like the python string method of the same name, which does a non-overlapping count!

For an overlapping search use the newer count_overlap() method.

Returns an integer, the number of occurrences of substring argument sub in the (sub)sequence given by [start:end]. Optional arguments start and end are interpreted as in slice notation.

Arguments:

- sub - a string or another Seq object to look for
- start - optional integer, slice start
- end - optional integer, slice end

e.g.

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AAAATGA")
>>> print(my_seq.count("A"))
5
>>> print(my_seq.count("ATG"))
1
>>> print(my_seq.count(Seq("AT")))
1
>>> print(my_seq.count("AT", 2, -1))
1
```

HOWEVER, please note because python strings and Seq objects (and MutableSeq objects) do a non-overlapping search, this may not give the answer you expect:

```
>>> "AAAA".count("AA")
2
>>> print(Seq("AAAA").count("AA"))
2
```

An overlapping search, as implemented in .count_overlap(), would give the answer as three!

count_overlap(*sub*, *start=0*, *end=9223372036854775807*)

Return an overlapping count.

For a non-overlapping search use the count() method.

Returns an integer, the number of occurrences of substring argument sub in the (sub)sequence given by [start:end]. Optional arguments start and end are interpreted as in slice notation.

Arguments:

- sub - a string or another Seq object to look for
- start - optional integer, slice start
- end - optional integer, slice end

e.g.

```
>>> from Bio.Seq import Seq
>>> print(Seq("AAAA").count_overlap("AA"))
3
>>> print(Seq("ATATATATA").count_overlap("ATA"))
4
>>> print(Seq("ATATATATA").count_overlap("ATA", 3, -1))
1
```

Where substrings do not overlap, should behave the same as the count() method:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AAAATGA")
>>> print(my_seq.count_overlap("A"))
5
>>> my_seq.count_overlap("A") == my_seq.count("A")
True
>>> print(my_seq.count_overlap("ATG"))
1
>>> my_seq.count_overlap("ATG") == my_seq.count("ATG")
True
>>> print(my_seq.count_overlap(Seq("AT")))
1
>>> my_seq.count_overlap(Seq("AT")) == my_seq.count(Seq("AT"))
True
>>> print(my_seq.count_overlap("AT", 2, -1))
1
>>> my_seq.count_overlap("AT", 2, -1) == my_seq.count("AT", 2, -1)
True
```

HOWEVER, do not use this method for such cases because the count() method is much for efficient.

endswith(*suffix*, *start=0*, *end=9223372036854775807*)

Return True if the Seq ends with the given suffix, False otherwise.

This behaves like the python string method of the same name.

Return True if the sequence ends with the specified suffix (a string or another Seq object), False otherwise. With optional start, test sequence beginning at that position. With optional end, stop comparing sequence at that position. suffix can also be a tuple of strings to try. e.g.

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.endswith("UUG")
True
>>> my_rna.endswith("AUG")
False
>>> my_rna.endswith("AUG", 0, 18)
True
>>> my_rna.endswith(("UCC", "UCA", "UUG"))
True
```

find(*sub*, *start=0*, *end=9223372036854775807*)
Find method, like that of a python string.

This behaves like the python string method of the same name.

Returns an integer, the index of the first occurrence of substring argument sub in the (sub)sequence given by [start:end].

Arguments:

- sub - a string or another Seq object to look for
- start - optional integer, slice start
- end - optional integer, slice end

Returns -1 if the subsequence is NOT found.

e.g. Locating the first typical start codon, AUG, in an RNA sequence:

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.find("AUG")
3
```

get_metadata(*label*, *only_first=False*)
Getter for the saved metadata with the key `label`

Parameters

- **label** (*str*) – key for the metadata that is inferred
- **only_first** (*bool*) – true if only the the first element of the matadata list is to be returned

log_metadata(*label*, *value*)
Inserts a new metadata

Parameters

- **label** (*str*) – key for the metadata that will be added
- **value** (*list(object)*) – any kindy of additional value that should be kept

lower()
Return a lower case copy of the sequence.

This will adjust the alphabet if required. Note that the IUPAC alphabets are upper case only, and thus a generic alphabet must be substituted.

```
>>> from Bio.Alphabet import Gapped, generic_dna
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
>>> my_seq = Seq("CGGTACGCTTATGTCACGTAG*AAAAAA",
...              Gapped(IUPAC.unambiguous_dna, "*"))
>>> my_seq
Seq('CGGTACGCTTATGTCACGTAG*AAAAAA', Gapped(IUPACUnambiguousDNA(), '*'))
>>> my_seq.lower()
Seq('cggtacgcttatgtcacgtag*aaaaaa', Gapped(DNAAlphabet(), '*'))
```

See also the upper method.

lstrip(*chars=None*)
Return a new Seq object with leading (left) end stripped.

This behaves like the python string method of the same name.

Optional argument chars defines which characters to remove. If omitted or None (default) then as for the python string method, this defaults to removing any white space.

e.g. print(my_seq.lstrip("-"))

See also the strip and rstrip methods.

newid = <method-wrapper 'next' of itertools.count object>

reverse_complement()
Return the reverse complement sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_dna = Seq("CCCCCGATAGNR", IUPAC.ambiguous_dna)
>>> my_dna
Seq('CCCCCGATAGNR', IUPACAmbiguousDNA())
>>> my_dna.reverse_complement()
Seq('YNCTATCGGGGG', IUPACAmbiguousDNA())
```

Note in the above example, since R = G or A, its complement is Y (which denotes C or T).

You can of course used mixed case sequences,

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("CCCCCgatA-G", generic_dna)
>>> my_dna
Seq('CCCCCgatA-G', DNAAlphabet())
>>> my_dna.reverse_complement()
Seq('C-TatcGGGGG', DNAAlphabet())
```

Trying to complement a protein sequence raises an exception:

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.reverse_complement()
Traceback (most recent call last):
   ...
ValueError: Proteins do not have complements!
```

rfind(*sub*, *start=0*, *end=9223372036854775807*)
Find from right method, like that of a python string.

This behaves like the python string method of the same name.

Returns an integer, the index of the last (right most) occurrence of substring argument sub in the (sub)sequence given by [start:end].

Arguments:

- sub - a string or another Seq object to look for
- start - optional integer, slice start
- end - optional integer, slice end

Returns -1 if the subsequence is NOT found.

e.g. Locating the last typical start codon, AUG, in an RNA sequence:

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.rfind("AUG")
15
```

rsplit(*sep=None*, *maxsplit=-1*)

Do a right split method, like that of a python string.

This behaves like the python string method of the same name.

Return a list of the 'words' in the string (as Seq objects), using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done COUNTING FROM THE RIGHT. If maxsplit is omitted, all splits are made.

Following the python string method, sep will by default be any white space (tabs, spaces, newlines) but this is unlikely to apply to biological sequences.

e.g. print(my_seq.rsplit("*",1))

See also the split method.

rstrip(*chars=None*)

Return a new Seq object with trailing (right) end stripped.

This behaves like the python string method of the same name.

Optional argument chars defines which characters to remove. If omitted or None (default) then as for the python string method, this defaults to removing any white space.

e.g. Removing a nucleotide sequence's polyadenylation (poly-A tail):

```
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
>>> my_seq = Seq("CGGTACGCTTATGTCACGTAGAAAAAA", IUPAC.unambiguous_dna)
>>> my_seq
Seq('CGGTACGCTTATGTCACGTAGAAAAAA', IUPACUnambiguousDNA())
>>> my_seq.rstrip("A")
Seq('CGGTACGCTTATGTCACGTAG', IUPACUnambiguousDNA())
```

See also the strip and lstrip methods.

split(*sep=None*, *maxsplit=-1*)

Split method, like that of a python string.

This behaves like the python string method of the same name.

Return a list of the 'words' in the string (as Seq objects), using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If maxsplit is omitted, all splits are made.

Following the python string method, sep will by default be any white space (tabs, spaces, newlines) but this is unlikely to apply to biological sequences.

e.g.

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_aa = my_rna.translate()
>>> my_aa
Seq('VMAIVMGR*KGAR*L', HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> for pep in my_aa.split("*"):
...     pep
Seq('VMAIVMGR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('KGAR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('L', HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> for pep in my_aa.split("*", 1):
...     pep
Seq('VMAIVMGR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('KGAR*L', HasStopCodon(ExtendedIUPACProtein(), '*'))
```

See also the rsplit method:

```
>>> for pep in my_aa.rsplit("*", 1):
...     pep
Seq('VMAIVMGR*KGAR', HasStopCodon(ExtendedIUPACProtein(), '*'))
Seq('L', HasStopCodon(ExtendedIUPACProtein(), '*'))
```

startswith(*prefix*, *start=0*, *end=9223372036854775807*)

Return True if the Seq starts with the given prefix, False otherwise.

This behaves like the python string method of the same name.

Return True if the sequence starts with the specified prefix (a string or another Seq object), False otherwise. With optional start, test sequence beginning at that position. With optional end, stop comparing sequence at that position. prefix can also be a tuple of strings to try. e.g.

```
>>> from Bio.Seq import Seq
>>> my_rna = Seq("GUCAUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAGUUG")
>>> my_rna.startswith("GUC")
True
>>> my_rna.startswith("AUG")
False
>>> my_rna.startswith("AUG", 3)
True
>>> my_rna.startswith(("UCC", "UCA", "UCG"), 1)
True
```

strip(*chars=None*)

Return a new Seq object with leading and trailing ends stripped.

This behaves like the python string method of the same name.

Optional argument chars defines which characters to remove. If omitted or None (default) then as for the python string method, this defaults to removing any white space.

e.g. print(my_seq.strip("-"))

See also the lstrip and rstrip methods.

tomutable()

Return the full sequence as a MutableSeq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("MKQHKAMIVALIVICITAVVAAL",
...              IUPAC.protein)
>>> my_seq
Seq('MKQHKAMIVALIVICITAVVAAL', IUPACProtein())
>>> my_seq.tomutable()
MutableSeq('MKQHKAMIVALIVICITAVVAAL', IUPACProtein())
```

Note that the alphabet is preserved.

tostring()
Return the full sequence as a python string (DEPRECATED).

You are now encouraged to use str(my_seq) instead of my_seq.tostring().

transcribe()
Return the RNA sequence from a DNA sequence by creating a new Seq object.

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG",
...                  IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> coding_dna.transcribe()
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
```

Trying to transcribe a protein or RNA sequence raises an exception:

```
>>> my_protein = Seq("MAIVMGR", IUPAC.protein)
>>> my_protein.transcribe()
Traceback (most recent call last):
   ...
ValueError: Proteins cannot be transcribed!
```

translate(*table='Standard'*, *stop_symbol='*'*, *to_stop=False*, *cds=False*, *gap=None*)
Turn a nucleotide sequence into a protein sequence by creating a new Seq object.

This method will translate DNA or RNA sequences, and those with a nucleotide or generic alphabet. Trying to translate a protein sequence raises an exception.

Arguments:

- table - Which codon table to use? This can be either a name (string), an NCBI identifier (integer), or a CodonTable object (useful for non-standard genetic codes). This defaults to the "Standard" table.
- stop_symbol - Single character string, what to use for terminators. This defaults to the asterisk, "*".
- to_stop - Boolean, defaults to False meaning do a full translation continuing on past any stop codons (translated as the specified stop_symbol). If True, translation is terminated at the first in frame stop codon (and the stop_symbol is not appended to the returned protein sequence).
- cds - Boolean, indicates this is a complete CDS. If True, this checks the sequence starts with a valid alternative start codon (which will be translated as methionine, M), that the sequence length is a multiple of three, and that there is a single in frame stop codon at the end (this will be excluded from the protein sequence, regardless of the to_stop option). If these tests fail, an exception is raised.

- gap - Single character string to denote symbol used for gaps. It will try to guess the gap character from the alphabet.

e.g. Using the standard table:

```
>>> coding_dna = Seq("GTGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
>>> coding_dna.translate()
Seq('VAIVMGR*KGAR*', HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> coding_dna.translate(stop_symbol="@")
Seq('VAIVMGR@KGAR@', HasStopCodon(ExtendedIUPACProtein(), '@'))
>>> coding_dna.translate(to_stop=True)
Seq('VAIVMGR', ExtendedIUPACProtein())
```

Now using NCBI table 2, where TGA is not a stop codon:

```
>>> coding_dna.translate(table=2)
Seq('VAIVMGRWKGAR*', HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> coding_dna.translate(table=2, to_stop=True)
Seq('VAIVMGRWKGAR', ExtendedIUPACProtein())
```

In fact, GTG is an alternative start codon under NCBI table 2, meaning this sequence could be a complete CDS:

```
>>> coding_dna.translate(table=2, cds=True)
Seq('MAIVMGRWKGAR', ExtendedIUPACProtein())
```

It isn't a valid CDS under NCBI table 1, due to both the start codon and also the in frame stop codons:

```
>>> coding_dna.translate(table=1, cds=True)
Traceback (most recent call last):
    ...
TranslationError: First codon 'GTG' is not a start codon
```

If the sequence has no in-frame stop codon, then the to_stop argument has no effect:

```
>>> coding_dna2 = Seq("TTGGCCATTGTAATGGGCCGC")
>>> coding_dna2.translate()
Seq('LAIVMGR', ExtendedIUPACProtein())
>>> coding_dna2.translate(to_stop=True)
Seq('LAIVMGR', ExtendedIUPACProtein())
```

When translating gapped sequences, the gap character is inferred from the alphabet:

```
>>> from Bio.Alphabet import Gapped
>>> coding_dna3 = Seq("GTG---GCCATT", Gapped(IUPAC.unambiguous_dna))
>>> coding_dna3.translate()
Seq('V-AI', Gapped(ExtendedIUPACProtein(), '-'))
```

It is possible to pass the gap character when the alphabet is missing:

```
>>> coding_dna4 = Seq("GTG---GCCATT")
>>> coding_dna4.translate(gap='-')
Seq('V-AI', Gapped(ExtendedIUPACProtein(), '-'))
```

NOTE - Ambiguous codons like "TAN" or "NNN" could be an amino acid or a stop codon. These are translated as "X". Any invalid codon (e.g. "TA?" or "T-A") will throw a TranslationError.

NOTE - This does NOT behave like the python string's translate method. For that use str(my_seq).translate(...) instead.

ungap(*gap=None*)

Return a copy of the sequence without the gap character(s).

The gap character can be specified in two ways - either as an explicit argument, or via the sequence's alphabet. For example:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("-ATA--TGAAAT-TTGAAAA", generic_dna)
>>> my_dna
Seq('-ATA--TGAAAT-TTGAAAA', DNAAlphabet())
>>> my_dna.ungap("-")
Seq('ATATGAAATTTGAAAA', DNAAlphabet())
```

If the gap character is not given as an argument, it will be taken from the sequence's alphabet (if defined). Notice that the returned sequence's alphabet is adjusted since it no longer requires a gapped alphabet:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC, Gapped, HasStopCodon
>>> my_pro = Seq("MVVLE=AD*", HasStopCodon(Gapped(IUPAC.protein, "=")))
>>> my_pro
Seq('MVVLE=AD*', HasStopCodon(Gapped(IUPACProtein(), '='), '*'))
>>> my_pro.ungap()
Seq('MVVLEAD*', HasStopCodon(IUPACProtein(), '*'))
```

Or, with a simpler gapped DNA example:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC, Gapped
>>> my_seq = Seq("CGGGTAG=AAAAAA", Gapped(IUPAC.unambiguous_dna, "="))
>>> my_seq
Seq('CGGGTAG=AAAAAA', Gapped(IUPACUnambiguousDNA(), '='))
>>> my_seq.ungap()
Seq('CGGGTAGAAAAAA', IUPACUnambiguousDNA())
```

As long as it is consistent with the alphabet, although it is redundant, you can still supply the gap character as an argument to this method:

```
>>> my_seq
Seq('CGGGTAG=AAAAAA', Gapped(IUPACUnambiguousDNA(), '='))
>>> my_seq.ungap("=")
Seq('CGGGTAGAAAAAA', IUPACUnambiguousDNA())
```

However, if the gap character given as the argument disagrees with that declared in the alphabet, an exception is raised:

```
>>> my_seq
Seq('CGGGTAG=AAAAAA', Gapped(IUPACUnambiguousDNA(), '='))
>>> my_seq.ungap("-")
Traceback (most recent call last):
  ...
ValueError: Gap '-' does not match '=' from alphabet
```

Finally, if a gap character is not supplied, and the alphabet does not define one, an exception is raised:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> my_dna = Seq("ATA--TGAAAT-TTGAAAA", generic_dna)
```

(continues on next page)

(continued from previous page)

```
>>> my_dna
Seq('ATA--TGAAAT-TTGAAAA', DNAAlphabet())
>>> my_dna.ungap()
Traceback (most recent call last):
   ...
ValueError: Gap character not given and not defined in alphabet
```

upper()
Return an upper case copy of the sequence.

```
>>> from Bio.Alphabet import HasStopCodon, generic_protein
>>> from Bio.Seq import Seq
>>> my_seq = Seq("VHLTPeeK*", HasStopCodon(generic_protein))
>>> my_seq
Seq('VHLTPeeK*', HasStopCodon(ProteinAlphabet(), '*'))
>>> my_seq.lower()
Seq('vhltpeek*', HasStopCodon(ProteinAlphabet(), '*'))
>>> my_seq.upper()
Seq('VHLTPEEK*', HasStopCodon(ProteinAlphabet(), '*'))
```

This will adjust the alphabet if required. See also the lower method.

1.1.8 Core.Variant

class Fred2.Core.Variant.**MutationSyntax**(*transID*, *transPos*, *protPos*, *cds*, *aas*, *geneID=None*)
This class represents the mutation syntax of a variant and stores its transcript and protein position

Parameters

- **transID** (*str*) – The *Transcript* id
- **transPos** (*int*) – The position of the *Variant* within the *Transcript*
- **protPos** (*int*) – The *Protein* position of the *Variant* within the *Transcript*
- **cds** (*str*) – The complete cds_mutation_syntax string
- **aas** (*str*) – The complete protein_mutation_syntax string

class Fred2.Core.Variant.**Variant**(*id*, *type*, *chrom*, *genomePos*, *ref*, *obs*, *coding*, *isHomozygous*, *isSynonymous*, *experimentalDesign=None*, *metadata=None*)
Bases: *Fred2.Core.Base.MetadataLogger*

A *Variant* contains information about a single genetic modification of the reference genome.

get_annotated_protein_pos(*transID*)
Returns the annotated protein position

Parameters **transID** (*str*) – The *Transcript* ID of interest

Returns The annotated *Protein* position of the given *Transcript* ID

Return type int

Raises **KeyError** – If *Variant* is not annotated to the given *Transcript* ID

get_annotated_transcript_pos(*transID*)
Returns the annotated *Transcript* position

Parameters **transID** (*str*) – The *Transcript* ID of interest

Returns The annotated *Transcript* position of the given *Transcript* ID

Return type int

Raises **KeyError** – If variant is not annotated to the given *Transcript* ID

get_metadata(*label*, *only_first=False*)
Getter for the saved metadata with the key `label`

Parameters

- **label** (*str*) – key for the metadata that is inferred
- **only_first** (*bool*) – true if only the the first element of the matadata list is to be returned

get_shift()
Returns the frameshift offset caused by the mutation in {0,1,2}

Returns The frameshift caused by mutation

Return type int

get_transcript_offset()
Returns the sequence offset caused by the mutation

Returns The sequence offset

Return type int

log_metadata(*label*, *value*)
Inserts a new metadata

Parameters

- **label** (*str*) – key for the metadata that will be added
- **value** (*list(object)*) – any kindy of additional value that should be kept

`Fred2.Core.Variant.`**VariationType**
alias of `Fred2.Core.Variant.Enum`

1.2 Fred2.IO Module

1.2.1 IO.ADBAdapter

1.2.2 IO.EnsemblAdapter

1.2.3 IO.FileReader

`Fred2.IO.FileReader.`**read_annovar_exonic**(*annovar_file*, *gene_filter=None*, *experimentalDesig=None*)
Reads an gene-based ANNOVAR output file and generates *Variant* objects containing all annotated *Transcript* ids an outputs a list *Variant*.

Parameters

- **annovar_file** (*str*) – The path ot the ANNOVAR file
- **gene_filter** (*list(str)*) – A list of gene names of interest (only variants associated with these genes are generated)

Returns List of :class:‘~Fred2.Core.Variant.Variants fully annotated

Return type list(*Variant*)

Fred2.IO.FileReader.**read_fasta**(*files*, *in_type=<class 'Fred2.Core.Peptide.Peptide'>*, *id_position=1*)

Generator function:

Read a (couple of) peptide, protein or rna sequence from a FASTA file. User needs to specify the correct type of the underlying sequences. It can either be: Peptide, Protein or Transcript (for RNA).

Parameters

- **files** – A (list) of file names to read in
- **in_type** (*Peptide* or *Transcript* or *Protein*) – The type to read in
- **id_position** (*int*) – the position of the id specified counted by |

In_type files list(str) or str

Returns a list of the specified sequence type derived from the FASTA file sequences.

Return type (list(in_type))

Raises **ValueError** – if a file is not readable

Fred2.IO.FileReader.**read_lines**(*files*, *in_type=<class 'Fred2.Core.Peptide.Peptide'>*)

Generator function:

Read a sequence directly from a line. User needs to manually specify the correct type of the underlying data. It can either be: Peptide, Protein or Transcript, Allele.

Parameters

- **files** – a list of strings of absolute file names that are to be read.
- **in_type** (*Peptide* or *Protein* or *Transcript* or *Allele*) – Possible in_type are *Peptide*, *Protein*, *Transcript*, and *Allele*.

In_type files list(str) or str

Returns A list of the specified objects

Return type (list(in_type))

Raises **IOError** – if a file is not readable

Fred2.IO.FileReader.**read_vcf**(*vcf_file*, *gene_filter=None*, *experimentalDesig=None*)

Reads an vcf v4.0 or 4.1 file and generates *Variant* objects containing all annotated *Transcript* ids an outputs a list *Variant*. Only the following variants are considered by the reader where synonymous labeled variants will not be integrated into any variant: filter_variants = [‘missense_variant’, ‘frameshift_variant’, ‘stop_gained’, ‘missense_variant&splice_region_variant’, “synonymous_variant”, “inframe_deletion”, “inframe_insertion”]

Parameters

- **vcf_file** (*str*) – The path ot the vcf file
- **gene_filter** (*list(str)*) – A list of gene names of interest (only variants associated with these genes are generated)

Returns List of :class:‘~Fred2.Core.Variant.Variants fully annotated

Return type Tuple of (list(*Variant*), list(transcript_ids)

1.2.4 IO.MartsAdapter

class Fred2.IO.MartsAdapter.**MartsAdapter**(*usr=None*, *host=None*, *pwd=None*, *db=None*, *biomart=None*)

Bases: Fred2.IO.ADBAdapter.ADBAdapter

get_all_variant_gene(*locations*, *_db='hsapiens_gene_ensembl'*, *_dataset='gene_ensembl_config'*)

Fetches the important db ids and names for given chromosomal location

Parameters

- **chrom** (*int*) – Integer value of the chromosome in question
- **start** (*int*) – Integer value of the variation start position on given chromosome
- **stop** (*int*) – Integer value of the variation stop position on given chromosome

Returns The respective gene name, i.e. the first one reported

get_all_variant_ids(***kwargs*)

Fetches the important db ids and names for given gene _or_ chromosomal location. The former is recommended. AResult is a list of dicts with either of the tree combinations:

- 'Ensembl Gene ID', 'Ensembl Transcript ID', 'Ensembl Protein ID'
- 'RefSeq Protein ID [e.g. NP_001005353]', 'RefSeq mRNA [e.g. NM_001195597]', first triplet
- 'RefSeq Predicted Protein ID [e.g. XP_001720922]', 'RefSeq mRNA predicted [e.g. XM_001125684]', first triplet

Parameters

- **'locations'** – list of locations as triplets of integer values representing (chrom, start, stop)
- **'genes'** – list of genes as string value of the genes of variation

Returns The list of dicts of entries with transcript and protein ids (either NM+NP or XM+XP)

get_ensembl_ids_from_id(*gene_id*, ***kwargs*)

Returns a list of gene-transcript-protein ids from some sort of id

Parameters

- **gene_id** (*str*) – The id to be queried
- **type** (EIdentifierTypes()) – Assumes given ID from type found in list of EIdentifierTypes() , default is gene name
- **_db** (*str*) – can override MartsAdapter default db ("hsapiens_gene_ensembl")
- **_dataset** (*str*) – specifies the query dbs dataset if default is not wanted ("gene_ensembl_config")

Returns Containing information about the corresponding (linked) entries.

Return type list(dict)

get_gene_by_position(*chrom*, *start*, *stop*, ***kwargs*)

Fetches the gene name for given chromosomal location

Parameters

- **chrom** (*int*) – Integer value of the chromosome in question

- **start** (*int*) – Integer value of the variation start position on given chromosome
- **stop** (*int*) – Integer value of the variation stop position on given chromosome
- **_db** (*str*) – Can override MartsAdapter default db ("hsapiens_gene_ensembl")
- **_dataset** (*str*) – Specifies the query dbs dataset if default is not wanted ("gene_ensembl_config")

Returns The respective gene name, i.e. the first one reported

Return type str

get_product_sequence(*product_id*, ***kwargs*)
Fetches product (i.e. protein) sequence for the given id

Parameters

- **product_id** (*str*) – The id to be queried
- **type** (EIdentifierTypes()) – Assumes given ID from type found in EIdentifierTypes(), default is ensembl_peptide_id
- **_db** (*str*) – Can override MartsAdapter default db ("hsapiens_gene_ensembl")
- **_dataset** (*str*) – Specifies the query dbs dataset if default is not wanted ("gene_ensembl_config")

Returns The requested sequence

Return type str

get_transcript_information(*transcript_id*, ***kwargs*)
Fetches transcript sequence, gene name and strand information for the given id

Parameters

- **transcript_id** (*str*) – The id to be queried
- **type** (EIdentifierTypes()) – Assumes given ID from type found in EIdentifierTypes(), default is ensembl_transcript_id
- **_db** (*str*) – Can override MartsAdapter default db ("hsapiens_gene_ensembl")
- **_dataset** (*str*) – Specifies the query dbs dataset if default is not wanted ("gene_ensembl_config")

Returns Dictionary of the requested keys as in EAdapterFields.ENUM

Return type dict

get_transcript_information_from_protein_id(*product_id*, ***kwargs*)
Fetches transcript sequence for the given id

Parameters

- **product_id** (*str*) – The id to be queried
- **type** (EIdentifierTypes()) – Assumes given ID from type found in EIdentifierTypes(), default is ensembl_peptide_id
- **_db** (*str*) – Can override MartsAdapter default db ("hsapiens_gene_ensembl")
- **_dataset** (*str*) – Specifies the query dbs dataset if default is not wanted ("gene_ensembl_config")

Returns List of dictionary of the requested sequence, the respective strand and the associated gene name

Return type list(dict)

get_transcript_position(*transcript_id*, *start*, *stop*, ***kwargs*)
If no transcript position is available for a variant, it can be retrieved if the mart has the transcripts connected to the CDS and the exons positions

Parameters

- **transcript_id** (*str*) – The id to be queried
- **start** (*int*) – First genomic position to be mapped
- **stop** (*int*) – Last genomic position to be mapped
- **type** (EIdentifierTypes()) – Assumes given ID from type found in EIdentifierTypes(), default is ensembl_transcript_id
- **_db** (*str*) – Can override MartsAdapter default db ("hsapiens_gene_ensembl")
- **_dataset** (*str*) – Specifies the query dbs dataset if default is not wanted ("gene_ensembl_config")

Returns A tuple of the mapped positions start, stop

Return type int

get_transcript_sequence(*transcript_id*, ***kwargs*)
Fetches transcript sequence for the given id

Parameters

- **transcript_id** (*str*) – The id to be queried
- **type** (EIdentifierTypes()) – Assumes given ID from type found in EIdentifierTypes(), default is ensembl_transcript_id
- **_db** (*str*) – Can override MartsAdapter default db ("hsapiens_gene_ensembl")
- **_dataset** (*str*) – Specifies the query dbs dataset if default is not wanted ("gene_ensembl_config")

Returns The requested sequence

Return type str

get_variant_id_from_protein_id(*transcript_id*, ***kwargs*)
Returns all information needed to instantiate a variation

Parameters

- **transcript_id** (*str*) – The id to be queried
- **type** (EIdentifierTypes()) – assumes given ID from type found in EIdentifierTypes(), default is ensembl_transcript_id
- **_db** (*str*) – can override MartsAdapter default db ("hsapiens_gene_ensembl")
- **_dataset** (*str*) – specifies the query dbs dataset if default is not wanted ("gene_ensembl_config")

Returns Containing all information needed for a variant initialization

Return type list(dict)

get_variant_ids(***kwargs*)
Fetches the important db ids and names for given gene _or_ chromosomal location. The former is recommended. AResult is a list of dicts with either of the tree combinations:

- 'Ensembl Gene ID', 'Ensembl Transcript ID', 'Ensembl Protein ID'
- 'RefSeq Protein ID [e.g. NP_001005353]', 'RefSeq mRNA [e.g. NM_001195597]', first triplet
- 'RefSeq Predicted Protein ID [e.g. XP_001720922]', 'RefSeq mRNA predicted [e.g. XM_001125684]', first triplet

Parameters

- **'chrom'** – integer value of the chromosome in question
- **'start'** – integer value of the variation start position on given chromosome
- **'stop'** – integer value of the variation stop position on given chromosome
- **'gene'** – string value of the gene of variation
- **'transcript_id'** – string value of the gene of variation

Returns The list of dicts of entries with transcript and protein ids (either NM+NP or XM+XP)

1.2.5 IO.RefSeqAdapter

Deprecated since version 1.0.

class Fred2.IO.RefSeqAdapter.**RefSeqAdapter**(***kwargs*)

Bases: Fred2.IO.ADBAdapter.ADBAdapter

get_product_sequence(*product_refseq*, ***kwargs*)

Fetches product sequence for the given id

Parameters **product_refseq** (*str*) – Given refseq id

Returns List of dictionaries of the requested sequence, the respective strand and the associated gene name

Return type list(dict)

get_transcript_information(*transcript_refseq*, ***kwargs*)

Fetches transcript sequence for the given id

Parameters

- **transcript_id** (*str*) – The transcript ID as string
- **type** – Given id, is in the form of this type,found in EIdentifierTypes(). It is to be documented if an ADBAdapter implementation overrides these types.

Returns list of dictionary of the requested sequence, the respective strand and the associated gene name

Return type list(dict)

get_transcript_sequence(*transcript_refseq*, ***kwargs*)

Fetches transcript sequence for the given id :param transcript_refseq: :return: list of dictionary of the requested sequence, the respective strand and the associated gene name

load(*filename*)

1.2.6 IO.UniProtAdapter

Deprecated since version 1.0.

class Fred2.IO.UniProtAdapter.**UniProtDB**(***kwargs*)

exists(*seq*)
fast check if given sequence exists (as subsequence) in one of the UniProtDB objects collection of sequences.

Parameters **seq** – the subsequence to be searched for

Returns True, if it is found somewhere, False otherwise

read_seqs(*sequence_file*)
read sequences from uniprot files (.dat or .fasta) or from lists or dicts of BioPython SeqRecords and make them available for fast search. Appending also with this function.

Parameters **sequence_file** – uniprot files (.dat or .fasta)

Returns

search(*seq*)
search for first occurrence of given sequence(s) in the UniProtDB objects collection returning (each) the fasta header front part of the first occurrence.

Parameters **seq** – a string interpreted as a single sequence or a list (of str) interpreted as a coll. of sequences

Returns a dictionary of sequences to lists (of ids, 'null' if n/a)

search_all(*seq*)
search for all occurrences of given sequence(s) in the UniProtDB objects collection returning (each) the fasta header front part of all occurrences.

Parameters **seq** – a string interpreted as a single sequence or a list (of str) interpreted as a coll. of sequences

Returns a dictionary of the given sequences to lists (of ids, 'null' if n/a)

write_seqs(*name*)
writes all fasta entries in the current object into one fasta file

Parameters **name** – the complete path with file name where the fasta is going to be written

CHAPTER 2

Prediction

2.1 Fred2.CleavagePrediction Module

2.1.1 CleavagePrediction.External

class Fred2.CleavagePrediction.External.**AExternalCleavageSitePrediction**

Bases: *Fred2.Core.Base.ACleavageSitePrediction*, *Fred2.Core.Base.AExternal*

Abstract base class for external cleavage site prediction methods. Implements predict functionality.

cleavagePos
Parameter specifying the position of aa (within the prediction window) after which the sequence is cleaved (starting from 1)

command
Defines the commandline call for external tool

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) –

- Optional specification of executable path if deviant from self.__command

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result (*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict (*aa_seq*, *command=None*, *options=None*, ***kwargs*)
Overwrites ACleavageSitePrediction.predict

Parameters

- **aa_seq** (list(*Peptide*/*Protein*) or *Peptide*/*Protein*) – A list of or a single *Peptide* or *Protein* object
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool

Returns A `CleavageSitePredictionResult` object

Return type `CleavageSitePredictionResult`

prepare_input (*input*, *file*)
Prepares the data :attr:_input and writes them to :attr:_file in the special format used by the external tool

Parameters

- **input** (*list(str)*) – The input data (here peptide sequences)
- **file** (*File*) – A file handler with which the data are written to file

supportedLength
The supported lengths of the predictor

version
Parameter specifying the version of the prediction method

class Fred2.CleavagePrediction.External.**NetChop_3_1**
Bases: *Fred2.CleavagePrediction.External.AExternalCleavageSitePrediction*, *Fred2.Core.Base.AExternal*

Implements NetChop Cleavage Site Prediction (v. 3.1).

Note: Nielsen, M., Lundegaard, C., Lund, O., & Kesmir, C. (2005). The role of the proteasome in generating cytotoxic T-cell epitopes: insights obtained from improved predictions of proteasomal cleavage. Immunogenetics, 57(1-2), 33-41.

---

cleavagePos
Parameter specifying the position of aa (within the prediction window) after which the sequence is cleaved

command
Defines the commandline call for external tool

get_external_version (*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) –

- Optional specification of executable path if deviant from :attr:self.__command

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*file*, *hash_dict*)
Parses external results and returns the result

Parameters

- **file** (*str*) – The file path or the external prediction results
- **hash_dict** (*dict(int,Protein)*) – A hash dict mapping from internal ID to Protein object du to ID length restriction of NetChop

Returns Returns a dictionary with the prediction results

Return type dict(str,dict((str,int),float))

predict(*aa_seq*, *command=None*, *options=None*, ***kwargs*)
Overwrites ACleavageSitePrediction.predict

Parameters

- **aa_seq** (list(*Peptide*/*Protein*) or *Peptide*/*Protein*) – A list of or a single *Peptide* or *Protein* object
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool

Returns A CleavageSitePredictionResult object

Return type CleavageSitePredictionResult

prepare_input(*input*, *file*)
Prepares the data and writes them to _file in the special format used by the external tool

Parameters

- **input** (*dict(int,Protein/Peptide)*) – The input data (here peptide sequences)
- **file** (*File*) – A file handler with which the data are written to file

supportedLength
The supported lengths of the predictor

version
The version of the Method

2.1.2 CleavagePrediction.PSSM

class Fred2.CleavagePrediction.PSSM.**APSSMCleavageFragmentPredictor**
Bases: *Fred2.Core.Base.ACleavageFragmentPrediction*

Abstract base class for PSSM predictions.

This implementation only supports cleavage fragment prediction not site prediction

Implements predict functionality

cleavagePos
Parameter specifying the position of aa (within the prediction window) after which the sequence is cleaved

name
The name of the predictor

predict (*peptides*, ***kwargs*)
Takes peptides plus their trailing C and N-terminal residues to predict the probability that this n-mer was produced by proteasomal cleavage. It returns the score and the peptide sequence in a AResult object. Row-IDs are the peitopes column is the prediction score.

Parameters **peptides** (list(*Peptide*) or *Peptide*) – A list of peptide objects or a single peptide object

Returns Returns a *Fred2.Core.Result.CleavageFragmentPredictionResult* object

Return type *Fred2.Core.Result.CleavageFragmentPredictionResult*

supportedLength
The supported lengths of the predictor

trailingN
The number of trailing residues at the N-terminal of the peptide used for prediction

tralingC
The number of trailing residues at the C-terminal of the peptide used for prediction

version
Parameter specifying the version of the prediction method

class Fred2.CleavagePrediction.PSSM.**APSSMCleavageSitePredictor**
Bases: *Fred2.Core.Base.ACleavageSitePrediction*

Abstract base class for PSSM predictions. This implementation only supports cleavage site prediction not fragment prediction. Implements predict functionality.

cleavagePos
Parameter specifying the position of aa (within the prediction window) after which the sequence is cleaved (starting from 1)

name
The name of the predictor

predict (*aa_seq*, *length=None*, ***kwargs*)
Returns predictions for given peptides.

Parameters

- **aa_seq** (list(*Peptide* or *Protein*) or *Peptide*/*Protein*) – A single *Peptide* or *~Fred2.Core.Protein.Protein* or a list of *Peptide* or *Protein*

- **length** (*int*) – The peptide length of the cleavage site model. If None the default value is used.

Returns Returns a *CleavageSitePredictionResult* object

Return type *CleavageSitePredictionResult*

supportedLength
The supported lengths of the predictor

version
Parameter specifying the version of the prediction method

class Fred2.CleavagePrediction.PSSM.**PCM**
Bases: *Fred2.CleavagePrediction.PSSM.APSSMCleavageSitePredictor*

Implements the PCM cleavage prediction method.

Note: Doennes, P., and Kohlbacher, O. (2005). Integrated modeling of the major events in the MHC class I antigen processing pathway. Protein Science, 14(8), 2132-2140.

---

cleavagePos
Parameter specifying the position of aa (within the prediction window) after which the sequence is cleaved

name
The name of the predictor

predict (*peptides*, *length=None*, ***kwargs*)
Returns predictions for given peptides.

Parameters

- **aa_seq** (list(*Peptide* or *Protein*) or *Peptide*/*Protein*) – A single *Peptide*/*~Fred2.Core.Protein.Protein* or a list of *Peptide*/*Protein*
- **length** (*int*) – The peptide length of the cleavage site model. If None the default value is used.

Returns Returns a *CleavageSitePredictionResult* object

Return type *CleavageSitePredictionResult*

supportedLength
A list of supported peptide lengths

version
The version of the predictor

class Fred2.CleavagePrediction.PSSM.**PSSMGinodi**
Bases: *Fred2.CleavagePrediction.PSSM.APSSMCleavageFragmentPredictor*

Implements the Cleavage Fragment prediction method of Ginodi et al.

Note: Ido Ginodi, Tal Vider-Shalit, Lea Tsaban, and Yoram Louzoun Precise score for the prediction of peptides cleaved by the proteasome Bioinformatics (2008) 24 (4): 477-483

---

cleavagePos
Parameter specifying the position of aa (within the prediction window) after which the sequence is cleaved

name
The name of the predictor

predict (*peptides*, ***kwargs*)
Takes peptides plus their trailing C and N-terminal residues to predict the probability that this n-mer was produced by proteasomal cleavage. It returns the score and the peptide sequence in a AResult object. Row-IDs are the peitopes column is the prediction score.

Parameters **peptides** (list(*Peptide*) or *Peptide*) – A list of peptide objects or a single peptide object

Returns Returns a *Fred2.Core.Result.CleavageFragmentPredictionResult* object

Return type *Fred2.Core.Result.CleavageFragmentPredictionResult*

supportedLength
A list of supported peptide lengths

trailingN
The number of trailing residues at the N-terminal of the peptide used for prediction

tralingC
The number of trailing residues at the C-terminal of the peptide used for prediction

version
The version of the predictor

class Fred2.CleavagePrediction.PSSM.**ProteaSMMConsecutive**
Bases: *Fred2.CleavagePrediction.PSSM.APSSMCleavageSitePredictor*

Implements the ProteaSMM cleavage prediction method.

Note: Tenzer, S., et al. "Modeling the MHC class I pathway by combining predictions of proteasomal cleavage, TAP transport and MHC class I binding." Cellular and Molecular Life Sciences CMLS 62.9 (2005): 1025-1037.

This model represents the consecutive proteasom

The matrices are generated not using the preon-dataset since a recent study has show that including those worsened the results.

Note: Calis, Jorg JA, et al. "Role of peptide processing predictions in T cell epitope identification: contribution of different prediction programs." Immunogenetics (2014): 1-9.

---

cleavagePos
Parameter specifying the position of aa (within the prediction window) after which the sequence is cleaved

name
The name of the predictor

predict (*peptides*, *length=None*, ***kwargs*)
Returns predictions for given peptides.

Parameters

- **aa_seq** (list(*Peptide* or *Protein*) or *Peptide*/*Protein*) – A single *Peptide*/*~Fred2.Core.Protein.Protein* or a list of *Peptide*/*Protein*
- **length** (*int*) – The peptide length of the cleavage site model. If None the default value is used.

Returns Returns a *CleavageSitePredictionResult* object

Return type *CleavageSitePredictionResult*

supportedLength
A list of supported peptide lengths

version
The version of the predictor

class Fred2.CleavagePrediction.PSSM.**ProteaSMMImmuno**
Bases: *Fred2.CleavagePrediction.PSSM.APSSMCleavageSitePredictor*

Implements the ProteaSMM cleavage prediction method.

Note: Tenzer, S., et al. "Modeling the MHC class I pathway by combining predictions of proteasomal cleavage, TAP transport and MHC class I binding." Cellular and Molecular Life Sciences CMLS 62.9 (2005): 1025-1037.

This model represents the immuno proteasom

The matrices are generated not using the preon-dataset since a recent study has show that including those worsened the results.

Note: Calis, Jorg JA, et al. "Role of peptide processing predictions in T cell epitope identification: contribution of different prediction programs." Immunogenetics (2014): 1-9.

---

cleavagePos
Parameter specifying the position of aa (within the prediction window) after which the sequence is cleaved

name
The name of the predictor

predict(*peptides*, *length=None*, ***kwargs*)
Returns predictions for given peptides.

Parameters

- **aa_seq** (list(*Peptide* or *Protein*) or *Peptide*/*Protein*) – A single *Peptide*/*~Fred2.Core.Protein.Protein* or a list of *Peptide*/*Protein*
- **length** (*int*) – The peptide length of the cleavage site model. If None the default value is used.

Returns Returns a *CleavageSitePredictionResult* object

Return type *CleavageSitePredictionResult*

supportedLength
A list of supported peptide lengths

version
The version of the predictor

2.1.3 Module contents

class Fred2.CleavagePrediction.**CleavageFragmentPredictorFactory**
Bases: object

static available_methods()
Returns a list of available cleavage site predictors

Returns dict(str,list(str)) - dict of cleavage site predictor represented as string and the supported versions

class Fred2.CleavagePrediction.**CleavageSitePredictorFactory**
Bases: object

static available_methods()
Returns a list of available cleavage site predictors

Returns dict(str,list(int)) - dict of cleavage site predictor represented as string and the supported versions

2.2 Fred2.TAPPrediction Module

2.2.1 TAPPrediction.PSSM

class Fred2.TAPPrediction.PSSM.**APSSMTAPPrediction**
Bases: *Fred2.Core.Base.ATAPPrediction*

Abstract base class for PSSM predictions. Implements predict functionality

name
The name of the predictor

predict(*peptides*, ***kwargs*)
Returns TAP predictions for given *Peptide*.

Parameters **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*

Returns Returns a *TAPPredictionResult* object with the prediction results

Return type *TAPPredictionResult*

supportedLength
The supported lengths of the predictor

version
Parameter specifying the version of the prediction method

class Fred2.TAPPrediction.PSSM.**SMMTAP**
Bases: *Fred2.TAPPrediction.PSSM.APSSMTAPPrediction*

Implementation of SMMTAP.

Note: Peters, B., Bulik, S., Tampe, R., Van Endert, P. M., & Holzhuetter, H. G. (2003). Identifying MHC class I epitopes by predicting the TAP transport efficiency of epitope precursors. The Journal of Immunology, 171(4), 1741-1749.

---

name
The name of the predictor

predict(*peptides*, ***kwargs*)
Returns TAP predictions for given *Peptide*.

Parameters **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*

Returns Returns a *TAPPredictionResult* object with the prediction results

Return type *TAPPredictionResult*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.TAPPrediction.PSSM.**TAPDoytchinova**
Bases: *Fred2.TAPPrediction.PSSM.APSSMTAPPrediction*

Implements the TAP prediction model from Doytchinova.

Note: Doytchinova, I., Hemsley, S. and Flower, D. R. Transporter associated with antigen processing preselection of peptides binding to the MHC: a bioinformatic evaluation. J. Immunol, 2004, 173, 6813-6819

---

name
The name of the predictor

predict (*peptides*, ***kwargs*)
Returns TAP predictions for given *Peptide*.

Parameters **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*

Returns Returns a *TAPPredictionResult* object with the prediction results

Return type *TAPPredictionResult*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

2.2.2 TAPPrediction.SVM

class Fred2.TAPPrediction.SVM.**ASVMTAPPrediction**
Bases: *Fred2.Core.Base.ATAPPrediction*, *Fred2.Core.Base.ASVM*

encode (*peptides*)
Returns the feature encoding for peptides

Parameters **peptides** (list(*Peptide*)/*Peptide*) – List of or a single *Peptide* object

Returns Feature encoding of the Peptide objects

Return type list(Object)

name
The name of the predictor

predict (*peptides*, ***kwargs*)
Returns TAP predictions for given *Peptide*.

Parameters **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*

Returns Returns a *TAPPredictionResult* object with the prediction results

Return type *TAPPredictionResult*

supportedLength
The supported lengths of the predictor

version
Parameter specifying the version of the prediction method

class Fred2.TAPPrediction.SVM.**SVMTAP**
Bases: *Fred2.TAPPrediction.SVM.ASVMTAPPrediction*

Implements SVMTAP prediction of Doeness et al.

Note: Doennes, P. and Kohlbacher, O. Integrated modeling of the major events in the MHC class I antigen processing pathway. Protein Sci, 2005

---

encode(*peptides*)
Encodes the *Peptide* with a binary sparse encoding

Parameters **peptides** (*list(str)*) – A list of *Peptide*

Returns Dictionary with *Peptide* as key and feature encoding as value (see svmlight encoding scheme http://svmlight.joachims.org/)

Return type dict(*Peptide*, (tuple(int, list(tuple(int,float))))

name
The name of the predictor

predict(*peptides*, ***kwargs*)
Returns predictions for given *Peptide*.

Parameters **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*

Returns Returns a *TAPPredictionResult* object with the prediction results

Return type *TAPPredictionResult*

supportedLength
A list of supported peptide lengths

version
The version of the predictor

2.2.3 Module contents

class Fred2.TAPPrediction.**TAPPredictorFactory**
Bases: object

static available_methods()
Returns a dictionary of available TAP predictors and the supported versions

Returns dict(str, list(str)) - A dictionary of TAP predictors represented as string and supported versions

2.3 Fred2.EpitopePrediction Module

2.3.1 EpitopePrediction.External

class Fred2.EpitopePrediction.External.**AExternalEpitopePrediction**
Bases: *Fred2.Core.Base.AEpitopePrediction*, *Fred2.Core.Base.AExternal*

Abstract class representing an external prediction function. Implementations shall wrap external binaries by following the given abstraction.

command
Defines the commandline call for external tool

convert_alleles(*alleles*)
Converts alleles into the internal allele representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The alleles for which the internal predictor representation is needed

Returns Returns a string representation of the input alleles

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) –

- Optional specification of executable path if deviant from self.__command

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict(*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method

- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input(*input*, *file*)
Prepares input for external tools and writes them to _file in the specific format

NO return value!

Param list(str) _input: The *Peptide* sequences to write into file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of valid allele models

supportedLength
A list of supported peptide lengths

version
The version of the predictor

class Fred2.EpitopePrediction.External.**NetCTLpan_1_1**
Bases: *Fred2.EpitopePrediction.External.AExternalEpitopePrediction*

Interface for NetCTLpan 1.1.

Note: NetCTLpan - Pan-specific MHC class I epitope predictions Stranzl T., Larsen M. V., Lundegaard C., Nielsen M. Immunogenetics. 2010 Apr 9. [Epub ahead of print]

---

command
Defines the commandline call for external tool

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (*Allele*) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `self.__command`

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict(*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input(*input*, *file*)
Prepares input for external tools and writes them to file in the specific format

No return value!

Param list(str) input: The *Peptide* sequences to write into file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.External.**NetMHCII_2_2**
Bases: *Fred2.EpitopePrediction.External.AExternalEpitopePrediction*

Implements a wrapper for NetMHCII

Note: Nielsen, M., & Lund, O. (2009). NN-align. An artificial neural network-based alignment algorithm for MHC class II peptide binding prediction. BMC Bioinformatics, 10(1), 296.

Nielsen, M., Lundegaard, C., & Lund, O. (2007). Prediction of MHC class II binding affinity using SMM-align, a novel stabilization matrix alignment method. BMC Bioinformatics, 8(1), 238.

---

command
Defines the commandline call for external tool

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (*Allele*) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `self.__command`

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict(*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input(*input*, *file*)
Prepares input for external tools and writes them to _file in the specific format

No return value!

Param list(str) input: The *Peptide* sequences to write into file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of valid *Allele* models

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.External.**NetMHCIIpan_3_0**
Bases: *Fred2.EpitopePrediction.External.AExternalEpitopePrediction*

Implements a wrapper for NetMHCIIpan.

Note: Andreatta, M., Karosiene, E., Rasmussen, M., Stryhn, A., Buus, S., & Nielsen, M. (2015). Accurate pan-specific prediction of peptide-MHC class II binding affinity with improved binding core identification. Immunogenetics, 1-10.

---

command
Defines the commandline call for external tool

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (*Allele*) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `self.__command`

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result (*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict (*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input (*input*, *file*)
Prepares input for external tools and writes them to _file in the specific format

No return value!

Param list(str) input: The *Peptide* sequences to write into file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of valid *Allele* models

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.External.**NetMHCIIpan_3_1**
Bases: *Fred2.EpitopePrediction.External.NetMHCIIpan_3_0*

Implementation of NetMHCIIpan 3.1 adapter.

Note: Andreatta, M., Karosiene, E., Rasmussen, M., Stryhn, A., Buus, S., & Nielsen, M. (2015). Accurate pan-specific prediction of peptide-MHC class II binding affinity with improved binding core identification. Immunogenetics, 1-10.

---

command
Defines the commandline call for external tool

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (*Allele*) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `self.__command`

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict(*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input(*input*, *file*)
Prepares input for external tools and writes them to _file in the specific format

No return value!

Param list(str) input: The *Peptide* sequences to write into file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of valid *Allele* models

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.External.**NetMHC_3_0**
Bases: *Fred2.EpitopePrediction.External.NetMHC_3_4*

Implements the NetMHC binding (for netMHC3.0):

```
.. note::
```

NetMHC-3.0: accurate web accessible predictions of human, mouse and monkey MHC class I affinities for peptides of length 8-11. Lundegaard C, Lamberth K, Harndahl M, Buus S, Lund O, Nielsen M. Nucleic Acids Res. 1;36(Web Server issue):W509-12. 2008

Accurate approximation method for prediction of class I MHC affinities for peptides of length 8, 10 and 11 using prediction tools trained on 9mers. Lundegaard C, Lund O, Nielsen M. Bioinformatics, 24(11):1397-98, 2008.

command
Defines the commandline call for external tool

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (*Allele*) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `self.__command`

Returns The external version of the tool or None if tool does not support versioning

Return type dict

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result (*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict (*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input (*input*, *file*)
Prepares input for external tools and writes them to file in the specific format

NO return value!

Param list(str) input: The : sequences to write into _file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of valid *Allele* models

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.External.**NetMHC_3_4**
Bases: *Fred2.EpitopePrediction.External.AExternalEpitopePrediction*

Implements the NetMHC binding (in current form for netMHC3.4).

Note: NetMHC-3.0: accurate web accessible predictions of human, mouse and monkey MHC class I affinities for peptides of length 8-11. Lundegaard C, Lamberth K, Harndahl M, Buus S, Lund O, Nielsen M. Nucleic Acids Res. 1;36(Web Server issue):W509-12. 2008

Accurate approximation method for prediction of class I MHC affinities for peptides of length 8, 10 and 11 using prediction tools trained on 9mers. Lundegaard C, Lund O, Nielsen M. Bioinformatics, 24(11):1397-98, 2008.

---

command
Defines the commandline call for external tool

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (*Allele*) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `self.__command`

Returns The external version of the tool or None if tool does not support versioning

Return type dict

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict(*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input (*input*, *file*)
Prepares input for external tools and writes them to file in the specific format

NO return value!

Param list(str) input: The : sequences to write into _file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of valid allele models

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.External.**NetMHC_4_0**
Bases: *Fred2.EpitopePrediction.External.NetMHC_3_4*

Implements the NetMHC 4.0 binding

Note: Andreatta M, Nielsen M. Gapped sequence alignment using artificial neural networks: application to the MHC class I system. Bioinformatics (2016) Feb 15;32(4):511-7

---

command
Defines the commandline call for external tool

convert_alleles (*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (*Allele*) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version (*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from self.__command

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result (*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict (*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input (*input*, *file*)
Prepares input for external tools and writes them to file in the specific format

NO return value!

Param list(str) input: The : sequences to write into _file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of valid allele models

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.External.**NetMHCpan_2_4**
Bases: *Fred2.EpitopePrediction.External.AExternalEpitopePrediction*

Implements the NetMHC binding (in current form for netMHCpan 2.4). Supported MHC alleles currently only restricted to HLA alleles.

Note: Nielsen, Morten, et al. "NetMHCpan, a method for quantitative predictions of peptide binding to any HLA-A and-B locus protein of known sequence." PloS one 2.8 (2007): e796.

---

command
Defines the commandline call for external tool

convert_alleles (*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (*Allele*) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `self.__command`

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict(*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input(*input*, *file*)
Prepares input for external tools and writes them to file in the specific format

NO return value!

Param list(str) input: The *Peptide* sequences to write into file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of valid *Allele* models

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.External.**NetMHCpan_2_8**
Bases: *Fred2.EpitopePrediction.External.AExternalEpitopePrediction*

Implements the NetMHC binding (in current form for netMHCpan 2.8). Supported MHC alleles currently only restricted to HLA alleles.

Note: Nielsen, Morten, et al. "NetMHCpan, a method for quantitative predictions of peptide binding to any HLA-A and-B locus protein of known sequence." PloS one 2.8 (2007): e796.

---

command
Defines the commandline call for external tool

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (*Allele*) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `self.__command`

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict (*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input (*input*, *file*)
Prepares input for external tools and writes them to file in the specific format

No return value!

Param list(str) input: The *Peptide* sequences to write into file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of valid *Allele* models

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.External.**NetMHCpan_3_0**
Bases: *Fred2.EpitopePrediction.External.NetMHCpan_2_8*

Implements the NetMHC binding version 3.0 Supported MHC alleles currently only restricted to HLA alleles.

Note: Nielsen, M., & Andreatta, M. (2016). NetMHCpan-3.0; improved prediction of binding to MHC class I molecules integrating information from multiple receptor and peptide length datasets. Genome Medicine, 8(1), 1.

---

command

convert_alleles (*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (*Allele*) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `self.__command`

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict(*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input(*input*, *file*)
Prepares input for external tools and writes them to file in the specific format

No return value!

Param list(str) input: The *Peptide* sequences to write into file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of valid *Allele* models

supportedLength
A list of supported *Peptide* lengths

version

class Fred2.EpitopePrediction.External.**NetMHCstabpan_1_0**
Bases: *Fred2.EpitopePrediction.External.AExternalEpitopePrediction*

Implements a wrapper to NetMHCstabpan 1.0

Pan-specific prediction of peptide-MHC-I complex stability; a correlate of T cell immunogenicity M Rasmussen, E Fenoy, M Nielsen, Buus S, Accepted JI June, 2016

command

convert_alleles(*alleles*)
Converts *Allele* into the internal allele representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `self.__command`

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name

parse_external_result(*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict(*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.

- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input(*input*, *file*)
Prepares input for external tools and writes them to file in the specific format

NO return value!

Param list(str) input: The *Peptide* sequences to write into file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles

supportedLength

version

class Fred2.EpitopePrediction.External.**PickPocket_1_1**
Bases: *Fred2.EpitopePrediction.External.AExternalEpitopePrediction*

Implementation of PickPocket adapter.

Note: Zhang, H., Lund, O., & Nielsen, M. (2009). The PickPocket method for predicting binding specificities for receptors based on receptor pocket similarities: application to MHC-peptide binding. Bioinformatics, 25(10), 1293-1299.

---

command
Defines the commandline call for external tool

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (*Allele*) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `elf.__command`

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result (*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict (*peptides*, *alleles=None*, *command=None*, *options=None*, ***kwargs*)
Overwrites AEpitopePrediction.predict

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A list of or a single *Peptide* object
- **alleles** (list(*Allele*)/*Allele*) – A list of or a single *Allele* object. If no *Allele* are provided, predictions are made for all *Allele* supported by the prediction method
- **command** (*str*) – The path to a alternative binary (can be used if binary is not globally executable)
- **options** (*str*) – A string of additional options directly past to the external tool.
- **chunksize** – denotes the chunksize in which the number of peptides are bulk processed

Returns A *EpitopePredictionResult* object

Return type *EpitopePredictionResult*

prepare_input (*input*, *file*)
Prepares input for external tools and writes them to file in the specific format

No return value!

Param list(str) input: The *Peptide* sequences to write into _file

Parameters **file** (*File*) – File-handler to input file for external tool

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

2.3.2 EpitopePrediction.PSSM

class Fred2.EpitopePrediction.PSSM.**APSSMEpitopePrediction**
Bases: *Fred2.Core.Base.AEpitopePrediction*

Abstract base class for PSSM predictions. Implements predict functionality

convert_alleles (*alleles*)
Converts alleles into the internal allele representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The alleles for which the internal predictor representation is needed

Returns Returns a string representation of the input alleles

Return type list(str)

name
The name of the predictor

predict (*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an *Allele*. If no *Allele* are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or class:*~Fred2.Core.Allele.Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of valid allele models

supportedLength
A list of supported peptide lengths

version
The version of the predictor

class Fred2.EpitopePrediction.PSSM.**ARB**
Bases: *Fred2.EpitopePrediction.PSSM.APSSMEpitopePrediction*

Implements IEDBs ARB method.

Note: Bui HH, Sidney J, Peters B, Sathiamurthy M, Sinichi A, Purton KA, Mothe BR, Chisari FV, Watkins DI, Sette A. 2005. Automated generation and evaluation of specific MHC binding predictive tools: ARB matrix applications. Immunogenetics 57:304-314.

---

convert_alleles (*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

name
The name of the predictor

predict (*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an *Allele*. If no *Allele* are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or class:*~Fred2.Core.Allele.Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.PSSM.**BIMAS**
Bases: *Fred2.EpitopePrediction.PSSM.APSSMEpitopePrediction*

Represents the BIMAS PSSM predictor.

Note: Parker, K.C., Bednarek, M.A. and Coligan, J.E. Scheme for ranking potential HLA-A2 binding peptides based on independent binding of individual peptide side-chains. The Journal of Immunology 1994;152(1):163-175.

---

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

name
The name of the predictor

predict(*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an *Allele*. If no *Allele* are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or class:*~Fred2.Core.Allele.Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.PSSM.**CalisImm**
Bases: *Fred2.EpitopePrediction.PSSM.APSSMEpitopePrediction*

Implements the Immunogenicity propensity score proposed by Calis et al.

..note:

Calis, Jorg JA, et al.(2013). Properties of MHC class I presented peptides that enhance immunogenicity. PLoS Comput Biol 9.10 e1003266.

convert_alleles(*alleles*)
Converts alleles into the internal allele representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The alleles for which the internal predictor representation is needed

Returns Returns a string representation of the input alleles

Return type list(str)

name
The name of the predictor

predict(*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an *Allele*. If no *Allele* are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **kwargs** – optional parameter (not used yet)

Returns Returns a `pandas.DataFrame` object with the prediction results

Return type `pandas.DataFrame`

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.PSSM.**ComblibSidney2008**
Bases: *Fred2.EpitopePrediction.PSSM.APSSMEpitopePrediction*

Implements IEDBs Comblib_Sidney2008 PSSM method.

Note: Sidney J, Assarsson E, Moore C, Ngo S, Pinilla C, Sette A, Peters B. 2008. Quantitative peptide binding motifs for 19 human and mouse MHC class I molecules derived using positional scanning combinatorial peptide libraries. Immunome Res 4:2.

---

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

name
The name of the predictor

predict (*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an *Allele*. If no *Allele* are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or class:*~Fred2.Core.Allele.Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.PSSM.**Epidemix**
Bases: *Fred2.EpitopePrediction.PSSM.APSSMEpitopePrediction*

Represents the Epidemix PSSM predictor.

Note: Feldhahn, M., et al. FRED-a framework for T-cell epitope detection. Bioinformatics 2009;25(20):2758-2759.

---

convert_alleles (*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

name
The name of the predictor

predict (*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an *Allele*. If no *Allele* are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or class:*~Fred2.Core.Allele.Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.PSSM.**Hammer**
Bases: *Fred2.EpitopePrediction.PSSM.APSSMEpitopePrediction*

Represents the virtual pockets approach by Sturniolo et al.

Note: Sturniolo, T., et al. Generation of tissue-specific and promiscuous HLA ligand databases using DNA microarrays and virtual HLA class II matrices. Nature biotechnology 1999;17(6):555-561.

---

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

name
The name of the predictor

predict(*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an *Allele*. If no *Allele* are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or class:*~Fred2.Core.Allele.Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.PSSM.**SMM**
Bases: *Fred2.EpitopePrediction.PSSM.APSSMEpitopePrediction*

Implements IEDBs SMM PSSM method.

Note: Peters B, Sette A. 2005. Generating quantitative models describing the sequence specificity of biological processes with the stabilized matrix method. BMC Bioinformatics 6:132.

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

name
The name of the predictor

predict(*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an *Allele*. If no *Allele* are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or class:*~Fred2.Core.Allele.Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.PSSM.**SMMPMBEC**
Bases: *Fred2.EpitopePrediction.PSSM.APSSMEpitopePrediction*

Implements IEDBs SMMPMBEC PSSM method.

Note: Kim, Y., Sidney, J., Pinilla, C., Sette, A., & Peters, B. (2009). Derivation of an amino acid similarity matrix for peptide: MHC binding and its application as a Bayesian prior. BMC Bioinformatics, 10(1), 394.

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

name
The name of the predictor

predict (*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an *Allele*. If no *Allele* are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or class:*~Fred2.Core.Allele.Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.PSSM.**Syfpeithi**
Bases: *Fred2.EpitopePrediction.PSSM.APSSMEpitopePrediction*

Represents the Syfpeithi PSSM predictor.

Note: Rammensee, H. G., Bachmann, J., Emmerich, N. P. N., Bachor, O. A., & Stevanovic, S. (1999). SYFPEITHI: database for MHC ligands and peptide motifs. Immunogenetics, 50(3-4), 213-219.

---

convert_alleles (*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

name
The name of the predictor

predict (*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an *Allele*. If no *Allele* are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or class:*~Fred2.Core.Allele.Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.PSSM.**TEPITOPEpan**
Bases: *Fred2.EpitopePrediction.PSSM.APSSMEpitopePrediction*

Implements TEPITOPEpan.

Note: TEPITOPEpan: Extending TEPITOPE for Peptide Binding Prediction Covering over 700 HLA-HLA-DR Molecules Zhang L, Chen Y, Wong H-S, Zhou S, Mamitsuka H, et al. (2012) TEPITOPEpan: Extending TEPITOPE for Peptide Binding Prediction Covering over 700 HLA-HLA-DR Molecules. PLoS ONE 7(2): e30483.

---

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

name
The name of the predictor

predict(*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an *Allele*. If no *Allele* are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or class:*~Fred2.Core.Allele.Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of supported *Allele*

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

2.3.3 EpitopePrediction.SVM

class Fred2.EpitopePrediction.SVM.**ASVMEpitopePrediction**
Bases: *Fred2.Core.Base.AEpitopePrediction*, *Fred2.Core.Base.ASVM*

Implements default prediction routine for SVM based epitope prediction tools

convert_alleles(*alleles*)
Converts alleles into the internal allele representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The alleles for which the internal predictor representation is needed

Returns Returns a string representation of the input alleles

Return type list(str)

encode(*peptides*)
Returns the feature encoding for peptides

Parameters **peptides** (list(*Peptide*)/*Peptide*) – List of or a single *Peptide* object

Returns Feature encoding of the Peptide objects

Return type list(Object)

name
The name of the predictor

predict(*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an alleles. If no alleles are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or *Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of valid allele models

supportedLength
A list of supported peptide lengths

version
The version of the predictor

class Fred2.EpitopePrediction.SVM.**SVMHC**
Bases: *Fred2.EpitopePrediction.SVM.ASVMEpitopePrediction*

Implements SVMHC epitope prediction for MHC-I alleles (SYFPEITHI models).

Note: Doennes, P. and Kohlbacher, O. SVMHC: a server for prediction of MHC-binding peptides. Nucleic Acids Res, 2006, 34, W194-W197

---

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

encode(*peptides*)
Encodes the input with binary sparse encoding of the *Peptide*

Parameters **peptides** (*str*) – A list of *Peptide* sequences

Returns Dictionary with *Peptide* as key and feature encoding as value (see svmlight encoding scheme http://svmlight.joachims.org/)

Return type dict(*Peptide*, (tuple(int, list(tuple(int,float))))

name
The name of the predictor

predict(*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an alleles. If no alleles are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or *Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of supported allele models

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

class Fred2.EpitopePrediction.SVM.**UniTope**
Bases: *Fred2.EpitopePrediction.SVM.ASVMEpitopePrediction*

Implements UniTope prediction for MHC-I.

Note: Toussaint, N. C., Feldhahn, M., Ziehm, M., Stevanovic, S., & Kohlbacher, O. (2011, August). T-cell epitope prediction based on self-tolerance. In Proceedings of the 2nd ACM Conference on Bioinformatics, Computational Biology and Biomedicine (pp. 584-588). ACM.

---

convert_alleles(*alleles*)
Converts *Allele* into the internal *Allele* representation of the predictor and returns a string representation

Parameters **alleles** (list(*Allele*)) – The *Allele* for which the internal predictor representation is needed

Returns Returns a string representation of the input *Allele*

Return type list(str)

encode(*peptides*, *allele*)
Encodes the input with binary sparse encoding of the *Peptide*

Parameters

- **peptides** (*str*) – A list of *Peptide* sequences
- **allele** (*str*) – The HLA *Allele* represented by a string

Returns Dictionary with *Peptide* as key and feature encoding as value (see svmlight encoding scheme http://svmlight.joachims.org/)

Return type dict(*Peptide*, (tuple(int, list(tuple(int,float))))‘

name
The name of the predictor

predict(*peptides*, *alleles=None*, ***kwargs*)
Returns predictions for given peptides an alleles. If no alleles are given, predictions for all available models are made.

Parameters

- **peptides** (list(*Peptide*) or *Peptide*) – A single *Peptide* or a list of *Peptide*
- **alleles** (list(*Allele*) or *Allele*) – A list of *Allele*
- **kwargs** – optional parameter (not used yet)

Returns Returns a *EpitopePredictionResult* object with the prediction results

Return type *EpitopePredictionResult*

supportedAlleles
A list of supported *Allele* models

supportedLength
A list of supported *Peptide* lengths

version
The version of the predictor

2.3.4 Module contents

CHAPTER 3

Vaccine Design

3.1 Fred2.EpitopeSelection Module

3.1.1 EpitopeSelection.OptiTope

class Fred2.EpitopeSelection.OptiTope.**OptiTope**(*results*, *threshold=None*, *k=10*, *solver='glpk'*, *verbosity=0*)

Bases: object

This class implements the epitope selection functionality of OptiTope published by Toussaint et al. [1].

This module builds upon Pyomo, an embedded algebraic modeling languages [2].

It allows to (de)select specific constraints of the underlying ILP and to solve the specific problem with a MIP solver of choice

Note: [1] N. C. Toussaint and O. Kohlbacher. OptiTope–a web server for the selection of an optimal set of peptides for epitope-based vaccines. Nucleic Acids Res, 2009, 37, W617-W622 [2] Pyomo - Optimization Modeling in Python. William E. Hart, Carl Laird, Jean-Paul Watson and David L. Woodruff. Springer, 2012.

activate_allele_coverage_const(*minCoverage*)

Enables the allele coverage constraint

Parameters **minCoverage** (*float*) – Percentage of alleles which have to be covered [0,1]

Raises **ValueError** – If the input variable is not in the same domain as the parameter

activate_antigen_coverage_const(*t_var*)

Activates the variation coverage constraint

Parameters **t_var** (*int*) – The number of epitopes which have to come from each variation

Raises **ValueError** – If the input variable is not in the same domain as the parameter

activate_epitope_conservation_const(*t_c*, *conservation=None*)

Activates the epitope conservation constraint

Parameters **t_c** (*float*) – The percentage of conservation an epitope has to have [0.0,1.0].

Param conservation: A dict with key=:class:~*Fred2.Core.Peptide.Peptide* specifying a different conservation score for each *Peptide*

Raises **ValueError** – If the input variable is not in the same domain as the parameter

deactivate_allele_coverage_const()
Deactivates the allele coverage constraint

deactivate_antigen_coverage_const()
Deactivates the variation coverage constraint

deactivate_epitope_conservation_const()
Deactivates epitope conservation constraint

set_k(*k*)
Sets the number of epitopes to select

Parameters **k** (*int*) – The number of epitopes

Raises **ValueError** – If the input variable is not in the same domain as the parameter

solve(*options=None*)
Invokes the selected solver and solves the problem

Parameters **options** (*dict(str,str)*) – A dictionary of solver specific options as keys and their parameters as values

:return Returns the optimal epitopes as list of *Peptide* objectives :rtype: list(*Peptide*) :raise RuntimeError: If the solver raised a problem or the solver is not accessible via the PATH

environmental variable.

3.2 Fred2.EpitopeAssembly Module

3.2.1 EpitopeAssembly.EpitopeAssembly

class Fred2.EpitopeAssembly.EpitopeAssembly.**EpitopeAssembly**(*peptides*, *pred*, *solver='glpk'*, *weight=0.0*, *matrix=None*, *verbosity=0*)

Bases: object

Implements the epitope assembly approach proposed by Toussaint et al. using proteasomal cleavage site prediction and formulating the problem as TSP.

Note: Toussaint, N.C., et al. Universal peptide vaccines - Optimal peptide vaccine design based on viral sequence conservation. Vaccine 2011;29(47):8745-8753.

Parameters

- **peptides** (list(*Peptide*)) – A list of *Peptide* which shell be arranged
- **pred** (ACleavageSitePredictor) – A *ACleavageSitePrediction*
- **solver** (*str*) – Specifies the solver to use (mused by callable by pyomo)

- **weight** (*float*) – Specifies how strong unwanted cleavage sites should be punished [0,1], where 0 means they will be ignored, and 1 the sum of all unwanted cleave sites is subtracted from the cleave site between two epitopes
- **verbosity** (*int*) – Specifies how verbos the class will be, 0 means normal, >0 debug mode

approximate()
Approximates the eptiope assembly problem by applying Lin-Kernighan traveling salesman heuristic

Note: LKH implementation must be downloaded, compiled, and globally executable. Source code can be found here: http://www.akira.ruc.dk/~keld/research/LKH/

Returns An order list of the *Peptide* (based on the sting-of-beads ordering)

Return type list(*Peptide*)

solve(*options=None*)
Solves the Epitope Assembly problem and returns an ordered list of the peptides

Note: This can take quite long and should not be done for more and 30 epitopes max!

Parameters **options** (*str*) – Solver specific options as string (will not be checked for correctness)

Returns An order list of the *Peptide* (based on the string-of-beads ordering)

Return type list(*Peptide*)

class Fred2.EpitopeAssembly.EpitopeAssembly.**EpitopeAssemblyWithSpacer**(*peptides*, *cleav_pred*, *epi_pred*, *alleles*, *k=5*, *en=9*, *threshold=None*, *solver='glpk'*, *alpha=0.99*, *beta=0*, *verbosity=0*)
Bases: object

Implements the epitope assembly approach proposed by Toussaint et al. using proteasomal cleavage site prediction and formulating the problem as TSP.

It also extends it by optimal spacer design. (currently only allowed with PSSM cleavage site and epitope prediction)

The ILP model is implemented. So be reasonable with the size of epitope to be arranged.

approximate(*start=0*, *threads=1*, *options=None*)

Approximates the Eptiope Assembly problem by applying Lin-Kernighan traveling salesman heuristic

LKH implementation must be downloaded, compiled, and globally executable.

Source code can be found here: http://www.akira.ruc.dk/~keld/research/LKH/

Parameters

- **start** (*int*) – Start length for spacers (default 0).
- **threads** (*int*) – Number of threads used for spacer design. Be careful, if options contain solver threads it will allocate threads*solver_threads cores!
- **options** (*dict(str,str)*) – Solver specific options (threads for example)

Returns A list of ordered *Peptide*

Return type list(*Peptide*)

solve(*start=0*, *threads=None*, *options=None*)

Solve the epitope assembly problem with spacers optimally using integer linear programming.

Note: This can take quite long and should not be done for more and 30 epitopes max! Also, one has to disable pre-solving steps in order to use this model.

Parameters

- **start** (*int*) – Start length for spacers (default 0).
- **threads** (*int*) – Number of threads used for spacer design. Be careful, if options contain solver threads it will allocate threads*solver_threads cores!
- **options** (*dict(str,str)*) – Solver specific options as keys and parameters as values

Returns A list of ordered *Peptide*

Return type list(*Peptide*)

class Fred2.EpitopeAssembly.EpitopeAssembly.**ParetoEpitopeAssembly**(*peptides*, *cl_pred*, *ep_pred*, *alleles*, *threshold*, *comparator*, *length=9*, *solver='glpk'*, *weight=0.0*, *matrix=None*, *verbosity=0*)

Bases: object

This implementation extends Toussaint et al.s TSP implementation which a bi-objective approach that also minimizes the neoepitope formation at the junctions as second objective. (Unpublished)

Note: Toussaint, N.C., et al. Universal peptide vaccines - Optimal peptide vaccine design based on viral sequence conservation. Vaccine 2011;29(47):8745-8753.

Parameters

- **peptides** (list(*Peptide*)) – A list of *Peptide* which shell be arranged
- **cl_pred** (ACleavageSitePredictor) – A *ACleavageSitePrediction*
- **ep_pred** (*AEpitopePrediction*) – A *AEpitopePrediction*
- **alleles** (List(*Allele*)) – A list of HLA alleles
- **threshold** (*dict(str,float)*) – a dictionary with key=allele.name and value the binding threshold of this allele
- **comparator** – A boolean function consuming two parameters a,b and comparing a comp b
- **length** (*int*) – the epitope length to consider (default: 9)
- **solver** (*str*) – Specifies the solver to use (mused by callable by pyomo)
- **weight** (*float*) – Specifies how strong unwanted cleavage sites should be punished [0,1], where 0 means they will be ignored, and 1 the sum of all unwanted cleave sites is subtracted from the cleave site between two epitopes
- **verbosity** (*int*) – Specifies how verbos the class will be, 0 means normal, >0 debug mode

paretosolve(*nof_sol=None*, *options={}*, *rel_tol=1e-09*, *abs_tol=0.0001*)
returns the whole pareto front of the be-objective optimization problem

Parameters

- **nof_sol** (*int*) – the number of solutions to max. obtain (if front is bigger)
- **options** (*dict*) – the solver options
- **rel_tol** (*float*) – relative floating point similarity tolerance
- **abs_tol** (*float*) – absolute floating point similarity tolerance

Returns the pareto front of possible assemblies

Return type list(tuple(float,float,list(*Peptide*)))

solve(*eps=1000000.0*, *order=(0, 1)*, *options={}*)
solves a bi-objective problem using the epsilon-constraint method

Parameters

- **options** (*str*) – options directly handed to the solver
- **eps** – the epsilon bound on the second objective
- **order** (*tuple(int,int)*) – The oder in which the two objective should be solved

Returns The two objective values and the pareot-optimal assembly as triple

Return type tuple(float,float,list(*Peptide*))

3.2.2 EpitopeAssembly.MosaicVaccine

CHAPTER 4

HLA Typing

4.1 Fred2.HLAtyping Module

4.1.1 HLAtyping.External

class Fred2.HLAtyping.External.**AExternalHLATyping**

Bases: *Fred2.Core.Base.AHLATyping*, *Fred2.Core.Base.AExternal*

clean_up(*_output*)

Cleans the generated files after prediction

Parameters **output** (*str*) – The path to the output file or directory

command

Defines the commandline call for external tool

get_external_version(*path=None*)

Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) –

- Optional specification of executable path if deviant from self.__command

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()

Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*file*)
Parses external results and returns the result

Parameters **file** (*str*) – The file path or the external prediction results

Returns A dictionary containing the prediction results

Return type dict

predict(*ngsFile*, *output*, *command=None*, *options=None*, *delete=True*, ***kwargs*)
Implementation of prediction

Parameters

- **ngsFile** (*str*) – The path to the NGS file of interest
- **output** (*str*) – The path to the output file or directory
- **command** (*str*) – The path to a alternative binary (if binary is not globally executable)
- **options** (*str*) – A string with additional options that is directly past to the tool
- **delete** (*bool*) – Boolean indicator whether generated files should be deleted afterwards

Returns A list of *Allele* objects representing the most likely HLA genotype

Return type list(*Allele*)

version
Parameter specifying the version of the prediction method

class Fred2.HLAtyping.External.**ATHLATES_1_0**
Bases: *Fred2.HLAtyping.External.AExternalHLATyping*

Wrapper for ATHLATES.

Note: C. Liu, X. Yang, B. Duffy, T. Mohanakumar, R.D. Mitra, M.C. Zody, J.D. Pfeifer (2012) ATHLATES: accurate typing of human leukocyte antigen through exome sequencing, Nucl. Acids Res. (2013)

---

clean_up(*output*)
Deletes files created by ATHLATES within _output

Parameters **output** (*str*) – The path to the output file or directory of the programme

command
Defines the commandline call for external tool

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from self.__command

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*output*)
Searches within the defined dir _file for the newest dir and reads the prediction file from there

Parameters **output** (*str*) – The path to the output dir

Returns The predicted HLA genotype

Return type list(*Allele*)

predict(*ngsFile*, *output*, *command=None*, *options=None*, *delete=True*, ***kwargs*)
Implementation of prediction

Parameters

- **ngsFile** (*str*) – The path to the NGS file of interest
- **output** (*str*) – The path to the output file or directory
- **command** (*str*) – The path to a alternative binary (if binary is not globally executable)
- **options** (*str*) – A string with additional options that is directly past to the tool
- **delete** (*bool*) – Boolean indicator whether generated files should be deleted afterwards

Returns A list of *Allele* objects representing the most likely HLA genotype

Return type list(*Allele*)

version
The version of the predictor

class Fred2.HLAtyping.External.**OptiType_1_0**
Bases: *Fred2.HLAtyping.External.AExternalHLATyping*

Wrapper of OptiType v1.0.

Note: Szolek, A., Schubert, B., Mohr, C., Sturm, M., Feldhahn, M., & Kohlbacher, O. (2014). OptiType: precision HLA typing from next-generation sequencing data. Bioinformatics, 30(23), 3310-3316.

---

clean_up(*output*)
Searches within the defined dir _file for the newest dir and deletes it. This should be the one OptiType had created

This could cause some terrible site effects if someone or something also writes in that directory!! OptiType should change the way it writes its output!

Parameters **output** (*str*) – The path to the output file or directory of the programme

command
Defines the commandline call for external tool

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from `self.command`

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*output*)
Searches within the defined dir _file for the newest dir and reads the prediction file from there

Parameters **output** (*str*) – The path to the output dir

Returns The predicted HLA genotype

Return type list(*Allele*)

predict(*ngsFile*, *output*, *command=None*, *options=None*, *delete=True*, ***kwargs*)
Implementation of prediction

Parameters

- **ngsFile** (*str*) – The path to the NGS file of interest
- **output** (*str*) – The path to the output file or directory
- **command** (*str*) – The path to a alternative binary (if binary is not globally executable)
- **options** (*str*) – A string with additional options that is directly past to the tool
- **delete** (*bool*) – Boolean indicator whether generated files should be deleted afterwards

Returns A list of *Allele* objects representing the most likely HLA genotype

Return type list(*Allele*)

version
The version of the predictor

class Fred2.HLAtyping.External.**Polysolver**
Bases: *Fred2.HLAtyping.External.AExternalHLATyping*

Wrapper for Polysolver.

Note: Shukla, Sachet A., Rooney, Michael S., Rajasagi, Mohini, Tiao, Grace, et al. (2015). Comprehensive analysis of cancer-associated somatic mutations in class I HLA genes. Nat Biotech, advance online publication. doi: 10.1038/nbt.3344

---

clean_up(*output*)
Deletes files created by Polysolver within output

Parameters **output** (*str*) – The path to the output file or directory of the programme

command
Defines the commandline call for external tool

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from self.__command

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*output*)
Searches within the defined dir _file for the newest dir and reads the prediction file from there

Parameters **output** (*str*) – The path to the output dir

Returns The predicted HLA genotype

Return type list(*Allele*)

predict(*ngsFile*, *output*, *command=None*, *options=None*, *delete=True*, ***kwargs*)
Implementation of prediction

Parameters

- **ngsFile** (*str*) – The path to the NGS file of interest
- **output** (*str*) – The path to the output file or directory
- **command** (*str*) – The path to a alternative binary (if binary is not globally executable)
- **options** (*str*) – A string with additional options that is directly past to the tool
- **delete** (*bool*) – Boolean indicator whether generated files should be deleted afterwards

Returns A list of *Allele* objects representing the most likely HLA genotype

Return type list(*Allele*)

version
The version of the predictor

class Fred2.HLAtyping.External.**Seq2HLA_2_2**
Bases: *Fred2.HLAtyping.External.AExternalHLATyping*

Wrapper of seq2HLA v2.2.

Note: Boegel, S., Scholtalbers, J., Loewer, M., Sahin, U., & Castle, J. C. (2015). In Silico HLA Typing Using Standard RNA-Seq Sequence Reads. Molecular Typing of Blood Cell Antigens, 247.

---

clean_up(*output*)
Deletes all created files.

Parameters **output** (*str*) – The path to the output file or directory of the programme

command
Defines the commandline call for external tool

get_external_version(*path=None*)
Returns the external version of the tool by executing >{command} –version

might be dependent on the method and has to be overwritten therefore it is declared abstract to enforce the user to overwrite the method. The function in the base class can be called with super()

Parameters **path** (*str*) – Optional specification of executable path if deviant from self.__command

Returns The external version of the tool or None if tool does not support versioning

Return type str

is_in_path()
Checks whether the specified execution command can be found in PATH

Returns Whether or not command could be found in PATH

Return type bool

name
The name of the predictor

parse_external_result(*output*)
Searches within the defined dir _file for the newest dir and reads the prediction file from there

Parameters **output** (*str*) – The path to the output dir

Returns The predicted HLA genotype

Return type list(*Allele*)

predict(*ngsFile*, *output*, *command=None*, *options=None*, *delete=True*, ***kwargs*)
Implementation of prediction

Parameters

- **ngsFile** (*str*) – The path to the NGS file of interest
- **output** (*str*) – The path to the output file or directory
- **command** (*str*) – The path to a alternative binary (if binary is not globally executable)
- **options** (*str*) – A string with additional options that is directly past to the tool
- **delete** (*bool*) – Boolean indicator whether generated files should be deleted afterwards

Returns A list of *Allele* objects representing the most likely HLA genotype

Return type list(*Allele*)

version
The version of the predictor

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X