ODDT Documentation

Release 0.4.1

Maciej Wojcikowski

Aug 09, 2017

Contents

Contents

CHAPTER 1

Installation

1.1 Requirements

- Python 2.7+ or 3.4+
- OpenBabel (2.3.2+) or/and RDKit (2016.03)
- Numpy (1.8+)
- Scipy (0.13+)
- Sklearn (0.18+)
- joblib (0.8+)
- pandas (0.13+)

Note: All installation methods assume that one of toolkits is installed. For detailed installation procedure visit toolkit's website (OpenBabel, RDKit)

Most convenient way of installing ODDT is using PIP. All required python modules will be installed automatically, although toolkits, either OpenBabel (`pip install openbabel`) or RDKit need to be installed manually

```
pip install oddt
```

If you want to install cutting edge version (master branch from GitHub) of ODDT also using PIP

```
pip install git+https://github.com/oddt/oddt.git@master
```

Finally you can install ODDT straight from the source

```
wget https://github.com/oddt/oddt/archive/0.4.1.tar.gz
tar zxvf 0.4.1.tar.gz
cd oddt-0.4.1/
python setup.py install
```

1.2 Common installation problems

CHAPTER 2

Usage Instructions

You can use any supported toolkit united under common API (for reference see Pybel or Cinfony). All methods and software which based on Pybel/Cinfony should be drop in compatible with ODDT toolkits. In contrast to it's predecessors, which were aimed to have minimalistic API, ODDT introduces extended methods and additional handles. This extensions allow to use toolkits at all it's grace and some features may be backported from others to introduce missing functionalities. To name a few:

- coordinates are returned as Numpy Arrays
- atoms and residues methods of Molecule class are lazy, ie. not returning a list of pointers, rather an object which allows indexing and iterating through atoms/residues
- Bond object (similar to Atom)
- *atom_dict*, *ring_dict*, *res_dict* - comprehensive Numpy Arrays containing common information about given entity, particularly useful for high performance computing, ie. interactions, scoring etc.
- lazy Molecule (asynchronous), which is not converted to an object in reading phase, rather passed as a string and read in when underlying object is called
- pickling introduced for Pybel Molecule (internally saved to mol2 string)

2.1 Atom, residues, bonds iteration

One of the most common operation would be iterating through molecules atoms

```
mol = oddt.toolkit.readstring('smi', 'c1cccc1')
for atom in mol:
    print(atom.idx)
```

Note: mol.atoms, returns an object (`AtomStack`) which can be access via indexes or iterated

Iterating over residues is also very convenient, especially for proteins

```
for res in mol.residues:
    print(res.name)
```

Additionally residues can fetch atoms belonging to them:

```
for res in mol.residues:
    for atom in res:
        print(atom.idx)
```

Bonds are also iterable, similar to residues:

```
for bond in mol.bonds:
    print(bond.order)
    for atom in bond:
        print(atom.idx)
```

2.2 Reading molecules

Reading molecules is mostly identical to Pybel.

Reading from file

```
for mol in oddt.toolkit.readfile('smi', 'test.smi'):
    print(mol.title)
```

Reading from string

```
mol = oddt.toolkit.readstring('smi', 'c1ccccc1 benzene'):
    print(mol.title)
```

Note: You can force molecules to be read in asynchronously, aka "lazy molecules". Current default is not to produce lazy molecules due to OpenBabel's Memory Leaks in OBConverter. Main advantage of lazy molecules is using them in multiprocessing, then conversion is spreaded on all jobs.

Reading molecules from file in asynchronous manner

```
for mol in oddt.toolkit.readfile('smi', 'test.smi', lazy=True):
    pass
```

This example will execute instantaneously, since no molecules were evaluated.

2.3 Numpy Dictionaries - store your molecule as an uniform structure

Most important and handy property of Molecule in ODDT are Numpy dictionaries containing most properties of supplied molecule. Some of them are straightforward, other require some calculation, ie. atom features. Dictionaries are provided for major entities of molecule: atoms, bonds, residues and rings. It was primarily used for interactions calculations, although it is applicable for any other calculation. The main benefit is marvelous Numpy broadcasting and subsetting.

Each dictionary is defined as a format in Numpy.

2.3.1 atom_dict

Atom basic information

- '*coords*', type: `float32`, shape: (3) - atom coordinates
- '*charge*', type: `float32` - atom's charge
- '*atomicnum*', type: `int8` - atomic number
- '*atomtype*', type: `a4` - Sybyl atom's type
- '*hybridization*', type: `int8` - atoms hybrydization
- '*neighbors*', type: `float32`, shape: (4,3) - coordinates of non-H neighbors coordinates for angles (max of 4 neighbors should be enough)

Residue information for current atom

- '*resid*', type: `int16` - residue ID
- '*resname*', type: `a3` - Residue name (3 letters)
- '*isbackbone*', type: `bool` - is atom part of backbone

Atom properties

- '*isacceptor*', type: `bool` - is atom H-bond acceptor
- '*isdonor*', type: `bool` - is atom H-bond donor
- '*isdonorh*', type: `bool` - is atom H-bond donor Hydrogen
- '*ismetal*', type: `bool` - is atom a metal
- '*ishydrophobe*', type: `bool` - is atom hydrophobic
- '*isaromatic*', type: `bool` - is atom aromatic
- '*isminus*', type: `bool` - is atom negatively charged/chargable
- '*isplus*', type: `bool` - is atom positively charged/chargable
- '*ishalogen*', type: `bool` - is atom a halogen

Secondary structure

- '*isalpha*', type: `bool` - is atom a part of alpha helix
- '*isbeta*', type: `bool'` - is atom a part of beta strand

2.3.2 ring_dict

- '*centroid*', type: `float32`, shape: 3 - coordinates of ring's centroid
- '*vector*', type: `float32`, shape: 3 - normal vector for ring
- '*isalpha*', type: `bool` - is ring a part of alpha helix
- '*isbeta*', type: `bool'` - is ring a part of beta strand

2.3.3 res_dict

- '*id*', type: `int16` - residue ID
- '*resname*', type: `a3` - Residue name (3 letters)
- '*N*', type: `float32`, shape: 3 - cordinates of backbone N atom
- '*CA*', type: `float32`, shape: 3 - cordinates of backbone CA atom
- '*C*', type: `float32`, shape: 3 - cordinates of backbone C atom
- '*isalpha*', type: `bool` - is residue a part of alpha helix
- '*isbeta*', type: `bool'` - is residue a part of beta strand

Note: All aforementioned dictionaries are generated "on demand", and are cached for molecule, thus can be shared between calculations. Caching of dictionaries brings incredible performance gain, since in some applications their generation is the major time consuming task.

Get all acceptor atoms:

```
mol.atom_dict['is_acceptor']
```

2.4 Interaction Fingerprints

Module, where interactions between two molecules are calculated and stored in fingerprint.

2.4.1 The most common usage

Firstly, loading files

```
protein = next(oddt.toolkit.readfile('pdb', 'protein.pdb'))
protein.protein = True
ligand = next(oddt.toolkit.readfile('sdf', 'ligand.sdf'))
```

Note: You have to mark a variable with file as protein, otherwise You won't be able to get access to e.g. 'resname; , 'resid' etc. It can be done as above.

File with more than one molecule

```
mols = list(oddt.toolkit.readfile('sdf', 'ligands.sdf'))
```

When files are loaded, You can check interactions between molecules. Let's find out, which amino acids creates hydrogen bonds

```
protein_atoms, ligand_atoms, strict = hbond(protein, ligand)
print(protein_atoms['resname'])
```

Or check hydrophobic contacts between molecules

```
protein_atoms, ligand_atoms = hydrophobic_contacts(protein, ligand)
print(protein_atoms, ligand_atoms)
```

But instead of checking interactions one by one, You can use fingerprints module.

```
IFP = InteractionFingerprint(ligand, protein)
SIFP = SimpleInteractionFingerprint(ligand, protein)
```

Very often we're looking for similar molecules. We can easily accomplish this by e.g.

```
results = []
reference = SimpleInteractionFingerprint(ligand, protein)
for el in query:
    fp_query = SimpleInteractionFingerprint(el, protein)
    # similarity score for current query
    cur_score = dice(reference, fp_query)
    # score is the lowest, required similarity
    if cur_score > score:
        results.append(el)
return results
```

2.5 Molecular shape comparison

Three methods for molecular shape comparison are supported: USR and its two derivatives: USRCAT and Electroshape.

- **USR (Ultrafast Shape Recognition) - function usr(molecule)** Ballester PJ, Richards WG (2007). Ultrafast shape recognition to search compound databases for similar molecular shapes. Journal of computational chemistry, 28(10):1711-23. http://dx.doi.org/10.1002/jcc.20681
- **USRCAT (USR with Credo Atom Types) - function usr_cat(molecule)** Adrian M Schreyer, Tom Blundell (2012). USRCAT: real-time ultrafast shape recognition with pharmacophoric constraints. Journal of Cheminformatics, 2012 4:27. http://dx.doi.org/10.1186/1758-2946-4-27
- **Electroshape - function electroshape(molecule)** Armstrong, M. S. et al. ElectroShape: fast molecular similarity calculations incorporating shape, chirality and electrostatics. J Comput Aided Mol Des 24, 789-801 (2010). http://dx.doi.org/doi:10.1007/s10822-010-9374-0

  Aside from spatial coordinates, atoms' charges are also used as the fourth dimension to describe shape of the molecule.

To find most similar molecules from the given set, each of these methods can be used.

Loading files:

```
query = next(oddt.toolkit.readfile('sdf', 'query.sdf'))
database = list(oddt.toolkit.readfile('sdf', 'database.sdf'))
```

Example code to find similar molecules:

```
results = []
query_shape = usr(query)
for mol in database:
    mol_shape = usr(mol)
    similarity = usr_similarity(query_shape, mol_shape)
    if similarity > 0.7:
        results.append(mol)
```

To use another method, replace usr(mol) with usr_cat(mol) or electroshape(mol).

CHAPTER 3

ODDT command line interface (CLI)

There is an *oddt* command to interface with Open Drug Discovery Toolkit from terminal, without any programming knowleadge. It simply reproduces *oddt.virtualscreening.virtualscreening*. One can filter, dock and score ligands using methods implemented or compatible with ODDT. All positional arguments are treated as input ligands, whereas output must be assigned using *-O* option (following *obabel* convention). Input and output formats are defined using *-i* and *-o* accordingly. If output format is present and no output file is assigned, then molecules are printed to STDOUT.

To list all the available options issue *-h* option:

```
oddt_cli -h
```

1. Docking ligand using Autodock Vina (construct box using ligand from crystal structure) with additional RFscore v2 rescoring:

```
oddt_cli input_ligands.sdf --dock autodock_vina --receptor rec.mol2 --auto_ligand␣
↪crystal_ligand.mol2 --score rfscore_v2 -O output_ligands.sdf
```

2. Filtering ligands using Lipinski RO5 and PAINS. Afterwards dock with Autodock Vina:

```
oddt_cli input_ligands.sdf --filter ro5 --filter pains --dock autodock_vina --
↪receptor rec.mol2 --auto_ligand crystal_ligand.mol2 -O output_ligands.sdf
```

3. Dock with Autodock Vina, with precise box position and dimensions. Fix seed for reproducibility and increase exhaustiveness:

```
oddt_cli ampc/actives_final.mol2.gz --dock autodock_vina --receptor ampc/receptor.pdb␣
↪--size '(8,8,8)' --center '(1,2,0.5)' --exhaustiveness 20 --seed 1 -O ampc_docked.
↪sdf
```

4. Rescore ligands using 3 versions of RFscore and pre-trained scoring function (either pickle from ODDT or any other SF implementing *oddt.scoring.scorer* API):

```
oddt_cli docked_ligands.sdf --receptor rec.mol2 --score rfscore_v1 --score rfscore_v2␣
↪--score rfscore_v3 --score TrainedNN.pickle -O docked_ligands_rescored.sdf
```

CHAPTER 4

Development and contributions guide

1. Indicies All indicies within toolkit are 0-based, but for backward compatibility with OpenBabel there is `mol.idx` property. If you develop using ODDT you are encouraged to use 0-based indicies and/or `mol.idx0` and `mol.idx1` properties to be exact which convention you adhere to. Otherwise you can run into bags which are hard to catch, when writing toolkit independent code.

CHAPTER 5

ODDT API documentation

5.1 oddt package

5.1.1 Subpackages

oddt.docking package

Submodules

oddt.docking.AutodockVina module

class oddt.docking.AutodockVina.**autodock_vina**(*protein=None*, *auto_ligand=None*, *size=(20, 20, 20)*, *center=(0, 0, 0)*, *exhaustiveness=8*, *num_modes=9*, *energy_range=3*, *seed=None*, *prefix_dir='/tmp'*, *n_cpu=1*, *executable=None*, *autocleanup=True*, *skip_bad_mols=True*)

Bases: object

Autodock Vina docking engine, which extends it's capabilities: automatic box (auto-centering on ligand).

Parameters **protein: oddt.toolkit.Molecule object (default=None)**

Protein object to be used while generating descriptors.

auto_ligand: oddt.toolkit.Molecule object or string (default=None) Ligand use to center the docking box. Either ODDT molecule or a file (opened based on extesion and read to ODDT molecule). Box is centered on geometric center of molecule.

size: tuple, shape=[3] (default=(20, 20, 20)) Dimentions of docking box (in Angstroms)

center: tuple, shape=[3] (default=(0,0,0)) The center of docking box in cartesian space.

exhaustiveness: int (default=8) Exhaustiveness parameter of Autodock Vina

num_modes: int (default=9) Number of conformations generated by Autodock Vina. The maximum number of docked poses is 9 (due to Autodock Vina limitation).

energy_range: int (default=3) Energy range cutoff for Autodock Vina

seed: int or None (default=None) Random seed for Autodock Vina

prefix_dir: string (default=/tmp) Temporary directory for Autodock Vina files

executable: string or None (default=None) Autodock Vina executable location in the system. It's realy necessary if autodetection fails.

autocleanup: bool (default=True) Should the docking engine clean up after execution?

skip_bad_mols: bool (default=True) Should molecules that crash Autodock Vina be skipped.

Attributes

tmp_dir

Methods

clean()	
dock(ligands[, protein, single])	Automated docking procedure.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands lazily
score(ligands[, protein, single])	Automated scoring procedure.
set_protein(protein)	Change protein to dock to.

clean()

dock(*ligands*, *protein=None*, *single=False*)

Automated docking procedure.

Parameters ligands: iterable of oddt.toolkit.Molecule objects

Ligands to dock

protein: oddt.toolkit.Molecule object or None Protein object to be used. If None, then the default one is used, else the protein is new default.

single: bool (default=False) A flag to indicate single ligand docking - performance reasons (eg. there is no need for subdirectory for one ligand)

Returns ligands : array of oddt.toolkit.Molecule objects

Array of ligands (scores are stored in mol.data method)

predict_ligand(*ligand*)

Local method to score one ligand and update it's scores.

Parameters ligand: oddt.toolkit.Molecule object

Ligand to be scored

Returns ligand: oddt.toolkit.Molecule object

Scored ligand with updated scores

predict_ligands(*ligands*)
Method to score ligands lazily

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to be scored

Returns ligand: iterator of oddt.toolkit.Molecule objects

Scored ligands with updated scores

score(*ligands*, *protein=None*, *single=False*)
Automated scoring procedure.

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to score

protein: oddt.toolkit.Molecule object or None Protein object to be used. If None, then the default one is used, else the protein is new default.

single: bool (default=False) A flag to indicate single ligand scoring - performance reasons (eg. there is no need for subdirectory for one ligand)

Returns **ligands** : array of oddt.toolkit.Molecule objects

Array of ligands (scores are stored in mol.data method)

set_protein(*protein*)
Change protein to dock to.

Parameters **protein: oddt.toolkit.Molecule object**

Protein object to be used.

tmp_dir

oddt.docking.AutodockVina.**parse_vina_docking_output**(*output*)
Function parsing Autodock Vina docking output to a dictionary

Parameters **output** : string

Autodock Vina standard ouptud (STDOUT).

Returns **out** : dict

dicitionary containing scores computed by Autodock Vina

oddt.docking.AutodockVina.**parse_vina_scoring_output**(*output*)
Function parsing Autodock Vina scoring output to a dictionary

Parameters **output** : string

Autodock Vina standard ouptud (STDOUT).

Returns **out** : dict

dicitionary containing scores computed by Autodock Vina

oddt.docking.internal module

ODDT's internal docking/scoring engines

oddt.docking.internal.**change_dihedral**(*coords*, *a1*, *a2*, *a3*, *a4*, *target_angle*, *rot_mask*)

oddt.docking.internal.**get_children**(*molecule*, *mother*, *restricted*)

oddt.docking.internal.**get_close_neighbors**(*molecule*, *a_idx*, *num_bonds=1*)

oddt.docking.internal.**num_rotors_pdbqt**(*lig*)

class oddt.docking.internal.**vina_docking**(*rec*, *lig=None*, *box=None*, *box_size=1.0*, *weights=None*)

Bases: object

Methods

correct_radius(atom_dict)
score([coords])
score_inter([coords])
score_intra([coords])
score_total([coords])
set_box(box)
set_coords(coords)
set_ligand(lig)
set_protein(rec)
weighted_inter([coords])
weighted_intra([coords])
weighted_total([coords])

correct_radius(*atom_dict*)

score(*coords=None*)

score_inter(*coords=None*)

score_intra(*coords=None*)

score_total(*coords=None*)

set_box(*box*)

set_coords(*coords*)

set_ligand(*lig*)

set_protein(*rec*)

weighted_inter(*coords=None*)

weighted_intra(*coords=None*)

weighted_total(*coords=None*)

class oddt.docking.internal.**vina_ligand**(*c0*, *num_rotors*, *engine*, *box_size=1*)

Bases: object

Methods

mutate(x2[, force])	

mutate (*x2*, *force=False*)

Module contents

class oddt.docking.**autodock_vina** (*protein=None*, *auto_ligand=None*, *size=(20, 20, 20)*, *center=(0, 0, 0)*, *exhaustiveness=8*, *num_modes=9*, *energy_range=3*, *seed=None*, *prefix_dir='/tmp'*, *n_cpu=1*, *executable=None*, *autocleanup=True*, *skip_bad_mols=True*)

Bases: object

Autodock Vina docking engine, which extends it's capabilities: automatic box (auto-centering on ligand).

Parameters **protein: oddt.toolkit.Molecule object (default=None)**

Protein object to be used while generating descriptors.

auto_ligand: oddt.toolkit.Molecule object or string (default=None) Ligand use to center the docking box. Either ODDT molecule or a file (opened based on extesion and read to ODDT molecule). Box is centered on geometric center of molecule.

size: tuple, shape=[3] (default=(20, 20, 20)) Dimentions of docking box (in Angstroms)

center: tuple, shape=[3] (default=(0,0,0)) The center of docking box in cartesian space.

exhaustiveness: int (default=8) Exhaustiveness parameter of Autodock Vina

num_modes: int (default=9) Number of conformations generated by Autodock Vina. The maximum number of docked poses is 9 (due to Autodock Vina limitation).

energy_range: int (default=3) Energy range cutoff for Autodock Vina

seed: int or None (default=None) Random seed for Autodock Vina

prefix_dir: string (default=/tmp) Temporary directory for Autodock Vina files

executable: string or None (default=None) Autodock Vina executable location in the system. It's realy necessary if autodetection fails.

autocleanup: bool (default=True) Should the docking engine clean up after execution?

skip_bad_mols: bool (default=True) Should molecules that crash Autodock Vina be skipped.

Attributes

tmp_dir	

Methods

clean()	
dock(ligands[, protein, single])	Automated docking procedure.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands lazily
score(ligands[, protein, single])	Automated scoring procedure.
set_protein(protein)	Change protein to dock to.

clean()

dock(*ligands*, *protein=None*, *single=False*)
Automated docking procedure.

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to dock

protein: oddt.toolkit.Molecule object or None Protein object to be used. If None, then the default one is used, else the protein is new default.

single: bool (default=False) A flag to indicate single ligand docking - performance reasons (eg. there is no need for subdirectory for one ligand)

Returns **ligands** : array of oddt.toolkit.Molecule objects

Array of ligands (scores are stored in mol.data method)

predict_ligand(*ligand*)
Local method to score one ligand and update it's scores.

Parameters **ligand: oddt.toolkit.Molecule object**

Ligand to be scored

Returns ligand: oddt.toolkit.Molecule object

Scored ligand with updated scores

predict_ligands(*ligands*)
Method to score ligands lazily

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to be scored

Returns ligand: iterator of oddt.toolkit.Molecule objects

Scored ligands with updated scores

score(*ligands*, *protein=None*, *single=False*)
Automated scoring procedure.

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to score

protein: oddt.toolkit.Molecule object or None Protein object to be used. If None, then the default one is used, else the protein is new default.

single: bool (default=False) A flag to indicate single ligand scoring - performance reasons (eg. there is no need for subdirectory for one ligand)

Returns **ligands** : array of oddt.toolkit.Molecule objects

Array of ligands (scores are stored in mol.data method)

set_protein(*protein*)
Change protein to dock to.

Parameters **protein: oddt.toolkit.Molecule object**

Protein object to be used.

tmp_dir

oddt.scoring package

Subpackages

oddt.scoring.descriptors package

Submodules

oddt.scoring.descriptors.binana module

Internal implementation of binana software (http://nbcr.ucsd.edu/data/sw/hosted/binana/)

class oddt.scoring.descriptors.binana.**binana_descriptor**(*protein=None*)
Bases: object

Descriptor build from binana script (as used in NNScore 2.0

Parameters **protein: oddt.toolkit.Molecule object (default=None)**

Protein object to be used while generating descriptors.

Methods

build(ligands[, protein])	Descriptor building method
set_protein(protein)	One function to change all relevant proteins

build(*ligands*, *protein=None*)
Descriptor building method

Parameters **ligands: array-like**

An array of generator of oddt.toolkit.Molecule objects for which the descriptor is computed

protein: oddt.toolkit.Molecule object (default=None) Protein object to be used while generating descriptors. If none, then the default protein (from constructor) is used. Otherwise, protein becomes new global and default protein.

Returns descs: numpy array, shape=[n_samples, 351]

An array of binana descriptors, aligned with input ligands

set_protein(*protein*)
One function to change all relevant proteins

Parameters **protein: oddt.toolkit.Molecule object**

Protein object to be used while generating descriptors. Protein becomes new global and default protein.

Module contents

class oddt.scoring.descriptors.**close_contacts**(*protein=None*, *cutoff=4*, *mode='atomic_nums'*, *ligand_types=None*, *protein_types=None*, *aligned_pairs=False*)
Bases: object

Close contacts descriptor which tallies atoms of type X in certain cutoff from atoms of type Y.

Parameters **protein: oddt.toolkit.Molecule or None (default=None)**

Default protein to use as reference

cutoff: int or list, shape=[n,] or shape=[n,2] (default=4) Cutoff for atoms in Angstroms given as an integer or a list of ranges, eg. [0, 4, 8, 12] or [[0,4],[4,8],[8,12]]. Upper bound is always inclusive, lower exclusive.

mode: string (default='atomic_nums') Method of atoms selection, as used in *atoms_by_type*

ligand_types: array List of ligand atom types to use

protein_types: array List of protein atom types to use

aligned_pairs: bool (default=False) Flag indicating should permutation of types should be done, otherwise the atoms are treated as aligned pairs.

Methods

build(ligands[, protein, single])	Builds descriptors for series of ligands

build(*ligands*, *protein=None*, *single=False*)
Builds descriptors for series of ligands

Parameters **ligands: iterable of oddt.toolkit.Molecules or oddt.toolkit.Molecule**

A list or iterable of ligands to build the descriptor or a single molecule.

protein: oddt.toolkit.Molecule or None (default=None) Default protein to use as reference

single: bool (default=False) Flag indicating if the ligand is single.

class oddt.scoring.descriptors.**fingerprints**(*fp='fp2'*, *toolkit='ob'*)
Bases: object

Methods

build(mols[, single])	

build(*mols*, *single=False*)

class oddt.scoring.descriptors.**autodock_vina_descriptor**(*protein=None*, *vina_scores=None*)

Bases: object

Methods

build(ligands[, protein, single])	
set_protein(protein)	

build(*ligands*, *protein=None*, *single=False*)

set_protein(*protein*)

class oddt.scoring.descriptors.**oddt_vina_descriptor**(*protein=None*, *vina_scores=None*)

Bases: object

Methods

build(ligands[, protein, single])	
set_protein(protein)	

build(*ligands*, *protein=None*, *single=False*)

set_protein(*protein*)

oddt.scoring.functions package

Submodules

oddt.scoring.functions.NNScore module

class oddt.scoring.functions.NNScore.**nnscore**(*protein=None*, *n_jobs=-1*)

Bases: *oddt.scoring.scorer*

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
gen_training_data(pdbbind_dir[, ...])	
load([filename, pdbbind_version])	
predict(ligands, *args, **kwargs)	Predicts values (eg.

Continued on next page

Table 5.12 – continued from previous page

predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands lazily
save(filename)	Saves scoring function to a pickle file.
score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction as squared correlation coefficient (R^2)
set_protein(protein)	Proxy method to update protein in all relevant places.
train([home_dir, sf_pickle, pdbbind_version])	

fit (*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

gen_training_data (*pdbbind_dir*, *pdbbind_versions=(2007*, *2012*, *2013*, *2014*, *2015*, *2016)*, *home_dir=None*)

classmethod load (*filename=''*, *pdbbind_version=2016*)

predict (*ligands*, **args*, ***kwargs*)
Predicts values (eg. affinity) for supplied ligands

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns predicted: np.array or array of np.arrays of shape = [n_ligands]

Predicted scores for ligands

predict_ligand (*ligand*)
Local method to score one ligand and update it's scores.

Parameters **ligand: oddt.toolkit.Molecule object**

Ligand to be scored

Returns ligand: oddt.toolkit.Molecule object

Scored ligand with updated scores

predict_ligands (*ligands*)
Method to score ligands lazily

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to be scored

Returns ligand: iterator of oddt.toolkit.Molecule objects

Scored ligands with updated scores

save (*filename*)
Saves scoring function to a pickle file.

Parameters **filename: string**

Pickle filename

score (*ligands*, *target*, **args*, ***kwargs*)
Methods estimates the quality of prediction as squared correlation coefficient (R^2)

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns r2: float

Squared correlation coefficient (R^2) for prediction

set_protein (*protein*)
Proxy method to update protein in all relevant places.

Parameters **protein: oddt.toolkit.Molecule object**

New default protein

train (*home_dir=None*, *sf_pickle=''*, *pdbbind_version=2016*)

oddt.scoring.functions.RFScore module

class oddt.scoring.functions.RFScore.**rfscore** (*protein=None*, *n_jobs=-1*, *version=1*, *spr=0*, ***kwargs*)
Bases: *oddt.scoring.scorer*

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
gen_training_data(pdbbind_dir[, ...])	
load([filename, version, pdbbind_version])	
predict(ligands, *args, **kwargs)	Predicts values (eg.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands lazily
save(filename)	Saves scoring function to a pickle file.
score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction as squared correlation coefficient (R^2)
set_protein(protein)	Proxy method to update protein in all relevant places.
train([home_dir, sf_pickle, pdbbind_version])	

fit (*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

gen_training_data(*pdbbind_dir*, *pdbbind_versions=(2007, 2012, 2013, 2014, 2015, 2016)*, *home_dir=None*)

classmethod load(*filename=''*, *version=1*, *pdbbind_version=2016*)

predict(*ligands*, **args*, ***kwargs*)

Predicts values (eg. affinity) for supplied ligands

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns predicted: np.array or array of np.arrays of shape = [n_ligands]

Predicted scores for ligands

predict_ligand(*ligand*)

Local method to score one ligand and update it's scores.

Parameters **ligand: oddt.toolkit.Molecule object**

Ligand to be scored

Returns ligand: oddt.toolkit.Molecule object

Scored ligand with updated scores

predict_ligands(*ligands*)

Method to score ligands lazily

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to be scored

Returns ligand: iterator of oddt.toolkit.Molecule objects

Scored ligands with updated scores

save(*filename*)

Saves scoring function to a pickle file.

Parameters **filename: string**

Pickle filename

score(*ligands*, *target*, **args*, ***kwargs*)

Methods estimates the quality of prediction as squared correlation coefficient (R^2)

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns r2: float

Squared correlation coefficient (R^2) for prediction

set_protein(*protein*)

Proxy method to update protein in all relevant places.

Parameters **protein: oddt.toolkit.Molecule object**

New default protein

train (*home_dir=None*, *sf_pickle=''*, *pdbbind_version=2016*)

Module contents

class oddt.scoring.functions.**rfscore** (*protein=None*, *n_jobs=-1*, *version=1*, *spr=0*, ***kwargs*)
Bases: *oddt.scoring.scorer*

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
gen_training_data(pdbbind_dir[, ...])	
load([filename, version, pdbbind_version])	
predict(ligands, *args, **kwargs)	Predicts values (eg.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands lazily
save(filename)	Saves scoring function to a pickle file.
score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction as squared correlation coefficient (R^2)
set_protein(protein)	Proxy method to update protein in all relevant places.
train([home_dir, sf_pickle, pdbbind_version])	

fit (*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

gen_training_data (*pdbbind_dir*, *pdbbind_versions=(2007, 2012, 2013, 2014, 2015, 2016)*, *home_dir=None*)

classmethod load (*filename=''*, *version=1*, *pdbbind_version=2016*)

predict (*ligands*, **args*, ***kwargs*)
Predicts values (eg. affinity) for supplied ligands

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns predicted: np.array or array of np.arrays of shape = [n_ligands]

Predicted scores for ligands

predict_ligand (*ligand*)
Local method to score one ligand and update it's scores.

Parameters **ligand: oddt.toolkit.Molecule object**

Ligand to be scored

Returns ligand: oddt.toolkit.Molecule object

Scored ligand with updated scores

predict_ligands(*ligands*)
Method to score ligands lazily

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to be scored

Returns ligand: iterator of oddt.toolkit.Molecule objects

Scored ligands with updated scores

save(*filename*)
Saves scoring function to a pickle file.

Parameters **filename: string**

Pickle filename

score(*ligands*, *target*, **args*, ***kwargs*)
Methods estimates the quality of prediction as squared correlation coefficient (R^2)

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns r2: float

Squared correlation coefficient (R^2) for prediction

set_protein(*protein*)
Proxy method to update protein in all relevant places.

Parameters **protein: oddt.toolkit.Molecule object**

New default protein

train(*home_dir=None*, *sf_pickle=''*, *pdbbind_version=2016*)

class oddt.scoring.functions.**nnscore**(*protein=None*, *n_jobs=-1*)
Bases: *oddt.scoring.scorer*

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
gen_training_data(pdbbind_dir[, ...])	
load([filename, pdbbind_version])	
predict(ligands, *args, **kwargs)	Predicts values (eg.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands lazily
save(filename)	Saves scoring function to a pickle file.

Continued on next page

Table 5.15 – continued from previous page

score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction as squared correlation coefficient (R^2)
set_protein(protein)	Proxy method to update protein in all relevant places.
train([home_dir, sf_pickle, pdbbind_version])	

fit (*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

gen_training_data (*pdbbind_dir*, *pdbbind_versions=(2007*, *2012*, *2013*, *2014*, *2015*, *2016)*, *home_dir=None*)

classmethod load (*filename=''*, *pdbbind_version=2016*)

predict (*ligands*, **args*, ***kwargs*)
Predicts values (eg. affinity) for supplied ligands

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns predicted: np.array or array of np.arrays of shape = [n_ligands]

Predicted scores for ligands

predict_ligand (*ligand*)
Local method to score one ligand and update it's scores.

Parameters **ligand: oddt.toolkit.Molecule object**

Ligand to be scored

Returns ligand: oddt.toolkit.Molecule object

Scored ligand with updated scores

predict_ligands (*ligands*)
Method to score ligands lazily

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to be scored

Returns ligand: iterator of oddt.toolkit.Molecule objects

Scored ligands with updated scores

save (*filename*)
Saves scoring function to a pickle file.

Parameters **filename: string**

Pickle filename

score (*ligands*, *target*, **args*, ***kwargs*)
Methods estimates the quality of prediction as squared correlation coefficient (R^2)

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns r2: float

Squared correlation coefficient (R^2) for prediction

set_protein (*protein*)
Proxy method to update protein in all relevant places.

Parameters **protein: oddt.toolkit.Molecule object**

New default protein

train (*home_dir=None*, *sf_pickle=''*, *pdbbind_version=2016*)

oddt.scoring.models package

Submodules

oddt.scoring.models.classifiers module

oddt.scoring.models.classifiers.**randomforest**
alias of RandomForestClassifier

class oddt.scoring.models.classifiers.**svm** (**args*, ***kwargs*)
Bases: sklearn.base.ClassifierMixin

Assemble a proper SVM classifier

Methods

fit(descs, target_values, **kwargs)
get_params([deep])
predict(descs)
predict_log_proba(descs)
predict_proba(descs)
score(descs, target_values)
set_params(**kwargs)

fit (*descs*, *target_values*, ***kwargs*)

get_params (*deep=True*)

predict (*descs*)

predict_log_proba (*descs*)

predict_proba (*descs*)

score (*descs*, *target_values*)

set_params (***kwargs*)

class oddt.scoring.models.classifiers.**neuralnetwork** (**args*, ***kwargs*)
Bases: sklearn.base.ClassifierMixin

Assemble Neural network using sklearn pipeline

Methods

fit(descs, target_values, **kwargs)
get_params([deep])
predict(descs)
predict_log_proba(descs)
predict_proba(descs)
score(descs, target_values)
set_params(**kwargs)

fit (*descs*, *target_values*, ***kwargs*)

get_params (*deep=True*)

predict (*descs*)

predict_log_proba (*descs*)

predict_proba (*descs*)

score (*descs*, *target_values*)

set_params (***kwargs*)

oddt.scoring.models.regressors module

Collection of regressors models

oddt.scoring.models.regressors.**randomforest**
alias of RandomForestRegressor

class oddt.scoring.models.regressors.**svm** (**args*, ***kwargs*)
Bases: sklearn.base.RegressorMixin

Assemble a proper SVM using sklearn tools regressor

Methods

fit(descs, target_values, **kwargs)
get_params([deep])
predict(descs)
score(descs, target_values)
set_params(**kwargs)

fit (*descs*, *target_values*, ***kwargs*)

get_params(*deep=True*)

predict(*descs*)

score(*descs*, *target_values*)

set_params(***kwargs*)

oddt.scoring.models.regressors.**pls**
alias of PLSRegression

class oddt.scoring.models.regressors.**neuralnetwork**(**args*, ***kwargs*)
Bases: sklearn.base.RegressorMixin

Assemble Neural network using sklearn pipeline

Methods

fit(descs, target_values, **kwargs)
get_params([deep])
predict(descs)
score(descs, target_values)
set_params(**kwargs)

fit(*descs*, *target_values*, ***kwargs*)

get_params(*deep=True*)

predict(*descs*)

score(*descs*, *target_values*)

set_params(***kwargs*)

oddt.scoring.models.regressors.**mlr**
alias of LinearRegression

Module contents

Module contents

oddt.scoring.**cross_validate**(*model*, *cv_set*, *cv_target*, *n=10*, *shuffle=True*, *n_jobs=1*)
Perform cross validation of model using provided data

Parameters **model: object**

Model to be tested

cv_set: array-like of shape = [n_samples, n_features] Estimated target values.

cv_target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

n: integer (default = 10) How many folds to be created from dataset

shuffle: bool (default = True) Should data be shuffled before folding.

n_jobs: integer (default = 1) How many CPUs to use during cross validation

Returns r2: array of shape = [n]

R^2 score for each of generated folds

class oddt.scoring.**ensemble_descriptor**(*descriptor_generators*)
Bases: object

Proxy class to build an ensemble of destriptors with an API as one

Parameters **models: array**

An array of models

Methods

build(mols, *args, **kwargs)	
set_protein(protein)	

build(*mols*, **args*, ***kwargs*)

set_protein(*protein*)

class oddt.scoring.**ensemble_model**(*models*)
Bases: object

Proxy class to build an ensemble of models with an API as one

Parameters **models: array**

An array of models

Methods

fit(X, y, *args, **kwargs)	
predict(X, *args, **kwargs)	
score(X, y, *args, **kwargs)	

fit(*X*, *y*, **args*, ***kwargs*)

predict(*X*, **args*, ***kwargs*)

score(*X*, *y*, **args*, ***kwargs*)

class oddt.scoring.**scorer**(*model_instance*, *descriptor_generator_instance*, *score_title='score'*)
Bases: object

Scorer class is parent class for scoring functions.

Parameters **model_instance: model**

Medel compatible with sklearn API (fit, predict and score methods)

descriptor_generator_instance: array of descriptors Descriptor generator object

score_title: string Title of score to be used.

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
load(filename)	Loads scoring function from a pickle file.
predict(ligands, *args, **kwargs)	Predicts values (eg.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands lazily
save(filename)	Saves scoring function to a pickle file.
score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction as squared correlation coefficient (R^2)
set_protein(protein)	Proxy method to update protein in all relevant places.

fit (*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

classmethod load (*filename*)
Loads scoring function from a pickle file.

Parameters **filename: string**

Pickle filename

Returns sf: scorer-like object

Scoring function object loaded from a pickle

predict (*ligands*, **args*, ***kwargs*)
Predicts values (eg. affinity) for supplied ligands

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns predicted: np.array or array of np.arrays of shape = [n_ligands]

Predicted scores for ligands

predict_ligand (*ligand*)
Local method to score one ligand and update it's scores.

Parameters **ligand: oddt.toolkit.Molecule object**

Ligand to be scored

Returns ligand: oddt.toolkit.Molecule object

Scored ligand with updated scores

predict_ligands (*ligands*)
Method to score ligands lazily

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to be scored

Returns ligand: iterator of oddt.toolkit.Molecule objects

Scored ligands with updated scores

save (*filename*)
Saves scoring function to a pickle file.

Parameters **filename: string**

Pickle filename

score (*ligands*, *target*, **args*, ***kwargs*)
Methods estimates the quality of prediction as squared correlation coefficient (R^2)

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns r2: float

Squared correlation coefficient (R^2) for prediction

set_protein (*protein*)
Proxy method to update protein in all relevant places.

Parameters **protein: oddt.toolkit.Molecule object**

New default protein

oddt.toolkits package

Subpackages

oddt.toolkits.extras package

Submodules

oddt.toolkits.extras.rdkit module

oddt.toolkits.extras.rdkit.**MolFromPDBBlock** (*molBlock*, *sanitize=True*, *removeHs=True*, *flavor=0*)

Module contents

Submodules

oddt.toolkits.common module

Code common to all toolkits

oddt.toolkits.common.**detect_secondary_structure**(*res_dict*)
Detect alpha helices and beta sheets in res_dict by phi and psi angles

oddt.toolkits.ob module

class oddt.toolkits.ob.**Atom**(*OBAtom*)
Bases: pybel.Atom

Attributes

atomicmass	
atomicnum	
bonds	
cidx	
coordidx	
coords	
exactmass	
formalcharge	
heavyvalence	
heterovalence	
hyb	
idx	DEPRECATED: RDKit is 0-based and OpenBabel is 1-based.
idx0	Note that this index is 0-based and OpenBabel's internal index in 1-based.
idx1	Note that this index is 1-based as OpenBabel's internal index.
implicitvalence	
isotope	
neighbors	
partialcharge	
residue	
spin	
type	
valence	
vector	

atomicmass

atomicnum

bonds

cidx

coordidx

coords

exactmass

formalcharge

heavyvalence

heterovalence

hyb

idx
DEPRECATED: RDKit is 0-based and OpenBabel is 1-based. State which convention you desire and use *idx0* or *idx1*.

Note that this index is 1-based as OpenBabel's internal index.

idx0
Note that this index is 0-based and OpenBabel's internal index in 1-based. Changed to be compatible with RDKit

idx1
Note that this index is 1-based as OpenBabel's internal index.

implicitvalence

isotope

neighbors

partialcharge

residue

spin

type

valence

vector

class oddt.toolkits.ob.**AtomStack**(*OBMol*)
Bases: object

class oddt.toolkits.ob.**Bond**(*OBBond*)
Bases: object

Attributes

atoms
isrotor
order

atoms

isrotor

order

class oddt.toolkits.ob.**BondStack**(*OBMol*)
Bases: object

class oddt.toolkits.ob.**Fingerprint**(*fingerprint*)
Bases: pybel.Fingerprint

Attributes

bits
raw

bits

raw

class oddt.toolkits.ob.**Molecule**(*OBMol=None*, *source=None*, *protein=False*)
Bases: pybel.Molecule

Attributes

OBMol	
atom_dict	
atoms	
bonds	
canonic_order	Returns np.array with canonic order of heavy atoms in the molecule
charge	
charges	
clone	
conformers	
coords	
data	
dim	
energy	
exactmass	
formula	
molwt	
num_rotors	Number of strict rotatable
res_dict	
residues	
ring_dict	
smiles	
spin	
sssr	
title	
unitcell	

Methods

addh([only_polar])	Add hydrogens
calccharges([model])	Estimates atomic partial charges in the molecule.
calcdesc([descnames])	Calculate descriptor values.
calcfp([fptype])	Calculate a molecular fingerprint.
clone_coords(source)	

Continued on next page

Table 5.27 – continued from previous page

convertdbonds()	Convert Dative Bonds.
draw([show, filename, update, usecoords])	Create a 2D depiction of the molecule.
localopt([forcefield, steps])	Locally optimize the coordinates.
make2D()	Generate 2D coordinates for molecule
make3D([forcefield, steps])	Generate 3D coordinates
removeh()	Remove hydrogens
write([format, filename, overwrite, opt, size])	

OBMol

addh(*only_polar=False*)
Add hydrogens

atom_dict

atoms

bonds

calccharges(*model='mmff94'*)
Estimates atomic partial charges in the molecule.

Optional parameters:

model – default is "mmff94". See the charges variable for a list of available charge models (in shell, *obabel -L charges*)

This method populates the *partialcharge* attribute of each atom in the molecule in place.

calcdesc(*descnames=[]*)
Calculate descriptor values.

Optional parameter: descnames – a list of names of descriptors

If descnames is not specified, all available descriptors are calculated. See the descs variable for a list of available descriptors.

calcfp(*fptype='FP2'*)
Calculate a molecular fingerprint.

Optional parameters:

fptype – the fingerprint type (default is "FP2"). See the fps variable for a list of of available fingerprint types.

canonic_order
Returns np.array with canonic order of heavy atoms in the molecule

charge

charges

clone

clone_coords(*source*)

conformers

convertdbonds()
Convert Dative Bonds.

coords

data

dim

draw(*show=True*, *filename=None*, *update=False*, *usecoords=False*)
Create a 2D depiction of the molecule.

Optional parameters: show – display on screen (default is True) filename – write to file (default is None) update – update the coordinates of the atoms to those

determined by the structure diagram generator (default is False)

usecoords – don't calculate 2D coordinates, just use the current coordinates (default is False)

Tkinter and Python Imaging Library are required for image display.

energy

exactmass

formula

localopt(*forcefield='mmff94'*, *steps=500*)
Locally optimize the coordinates.

Optional parameters:

forcefield – default is "mmff94". See the forcefields variable for a list of available forcefields.

steps – default is 500

If the molecule does not have any coordinates, make3D() is called before the optimization. Note that the molecule needs to have explicit hydrogens. If not, call addh().

make2D()
Generate 2D coordinates for molecule

make3D(*forcefield='mmff94'*, *steps=50*)
Generate 3D coordinates

molwt

num_rotors
Number of strict rotatable

removeh()
Remove hydrogens

res_dict

residues

ring_dict

smiles

spin

sssr

title

unitcell

write(*format='smi'*, *filename=None*, *overwrite=False*, *opt=None*, *size=None*)

class oddt.toolkits.ob.**MoleculeData**(*obmol*)
Bases: pybel.MoleculeData

Methods

clear()	
has_key(key)	
items()	
iteritems()	
keys()	
to_dict()	
update(dictionary)	
values()	

clear()

has_key(*key*)

items()

iteritems()

keys()

to_dict()

update(*dictionary*)

values()

class oddt.toolkits.ob.**Outputfile**(*format*, *filename*, *overwrite=False*, *opt=None*)
Bases: pybel.Outputfile

Methods

close()	Close the Outputfile to further writing.
write(molecule)	Write a molecule to the output file.

close()
Close the Outputfile to further writing.

write(*molecule*)
Write a molecule to the output file.

Required parameters: molecule

class oddt.toolkits.ob.**Residue**(*OBResidue*)
Bases: object

Represent a Pybel residue.

Required parameter: OBResidue – an Open Babel OBResidue

Attributes: atoms, idx, name.

(refer to the Open Babel library documentation for more info).

The original Open Babel atom can be accessed using the attribute: OBResidue

Attributes

atoms
idx
name

atoms

idx

name

class oddt.toolkits.ob.**ResidueStack**(*OBMol*)
Bases: object

class oddt.toolkits.ob.**Smarts**(*smartspattern*)
Bases: pybel.Smarts

Initialise with a SMARTS pattern.

Methods

findall(molecule[, unique])	Find all matches of the SMARTS pattern to a particular molecule
match(molecule)	Checks if there is any match.

findall(*molecule*, *unique=True*)
Find all matches of the SMARTS pattern to a particular molecule

match(*molecule*)
Checks if there is any match. Returns True or False

oddt.toolkits.ob.**readfile**(*format*, *filename*, *opt=None*, *lazy=False*)

oddt.toolkits.rdk module

rdkit - A Cinfony module for accessing the RDKit from CPython

Global variables: Chem and AllChem - the underlying RDKit Python bindings informats - a dictionary of supported input formats outformats - a dictionary of supported output formats descs - a list of supported descriptors fps - a list of supported fingerprint types forcefields - a list of supported forcefields

class oddt.toolkits.rdk.**Atom**(*Atom*)
Bases: object

Represent an rdkit Atom.

Required parameters: Atom – an RDKit Atom

Attributes: atomicnum, coords, formalcharge

The original RDKit Atom can be accessed using the attribute: Atom

Attributes

atomicnum	
bonds	
coords	
formalcharge	
idx	DEPRECATED: RDKit is 0-based and OpenBabel is 1-based.
idx0	Note that this index is 0-based as RDKit's
idx1	Note that this index is 1-based and RDKit's internal index in 0-based.
neighbors	
partialcharge	

atomicnum

bonds

coords

formalcharge

idx
DEPRECATED: RDKit is 0-based and OpenBabel is 1-based. State which convention you desire and use *idx0* or *idx1*.

Note that this index is 1-based and RDKit's internal index in 0-based. Changed to be compatible with OpenBabel

idx0
Note that this index is 0-based as RDKit's

idx1
Note that this index is 1-based and RDKit's internal index in 0-based. Changed to be compatible with OpenBabel

neighbors

partialcharge

class oddt.toolkits.rdk.**AtomStack**(*Mol*)
Bases: object

class oddt.toolkits.rdk.**Bond**(*Bond*)
Bases: object

Attributes

atoms	
isrotor	
order	

atoms

isrotor

order

class oddt.toolkits.rdk.**BondStack**(*Mol*)
Bases: object

class oddt.toolkits.rdk.**Fingerprint**(*fingerprint*)
Bases: object

A Molecular Fingerprint.

Required parameters: fingerprint – a vector calculated by one of the fingerprint methods

Attributes: fp – the underlying fingerprint object bits – a list of bits set in the Fingerprint

Methods: The "|" operator can be used to calculate the Tanimoto coeff. For example, given two Fingerprints 'a', and 'b', the Tanimoto coefficient is given by:

tanimoto = a | b

Attributes

raw	

raw

class oddt.toolkits.rdk.**Molecule**(*Mol=None*, *source=None*, *protein=False*)
Bases: object

Trap RDKit molecules which are 'None'

Attributes

Mol	
atom_dict	
atoms	
bonds	
canonic_order	Returns np.array with canonic order of heavy atoms in the molecule
charges	
clone	
coords	
data	
formula	
molwt	
num_rotors	
res_dict	
residues	
ring_dict	
smiles	
sssr	
title	

Methods

addh([only_polar])	Add hydrogens.
calcdesc([descnames])	Calculate descriptor values.
calcfp([fptype, opt])	Calculate a molecular fingerprint.
clone_coords(source)	
localopt([forcefield, steps])	Locally optimize the coordinates.
make2D()	Generate 2D coordinates for molecule
make3D([forcefield, steps])	Generate 3D coordinates.
removeh(**kwargs)	Remove hydrogens.
write([format, filename, overwrite, size])	Write the molecule to a file or return a string.

Mol

addh(*only_polar=False*, ***kwargs*)
Add hydrogens.

atom_dict

atoms

bonds

calcdesc(*descnames=None*)
Calculate descriptor values.

Optional parameter: descnames – a list of names of descriptors

If descnames is not specified, all available descriptors are calculated. See the descs variable for a list of available descriptors.

calcfp(*fptype='rdkit'*, *opt=None*)
Calculate a molecular fingerprint.

Optional parameters:

fptype – the fingerprint type (default is "rdkit"). See the fps variable for a list of of available fingerprint types.

opt – a dictionary of options for fingerprints. Currently only used for radius and bitInfo in Morgan fingerprints.

canonic_order
Returns np.array with canonic order of heavy atoms in the molecule

charges

clone

clone_coords(*source*)

coords

data

formula

localopt(*forcefield='uff'*, *steps=500*)
Locally optimize the coordinates.

Optional parameters:

forcefield – default is "uff". See the forcefields variable for a list of available forcefields.

steps – default is 500

If the molecule does not have any coordinates, make3D() is called before the optimization.

make2D()
Generate 2D coordinates for molecule

make3D(*forcefield='mmff94'*, *steps=50*)
Generate 3D coordinates.

Optional parameters:

forcefield – default is "uff". See the forcefields variable for a list of available forcefields.

steps – default is 50

Once coordinates are generated, a quick local optimization is carried out with 50 steps and the UFF forcefield. Call localopt() if you want to improve the coordinates further.

molwt

num_rotors

removeh(***kwargs*)
Remove hydrogens.

res_dict

residues

ring_dict

smiles

sssr

title

write(*format='smi'*, *filename=None*, *overwrite=False*, *size=None*, ***kwargs*)
Write the molecule to a file or return a string.

Optional parameters:

format – see the informats variable for a list of available output formats (default is "smi")

filename – default is None overwite – if the output file already exists, should it

be overwritten? (default is False)

If a filename is specified, the result is written to a file. Otherwise, a string is returned containing the result.

To write multiple molecules to the same file you should use the Outputfile class.

class oddt.toolkits.rdk.**MoleculeData**(*Mol*)
Bases: object

Store molecule data in a dictionary-type object

Required parameters: Mol – an RDKit Mol

Methods and accessor methods are like those of a dictionary except that the data is retrieved on-the-fly from the underlying Mol.

Example: >>> mol = next(readfile("sdf", 'head.sdf')) >>> data = mol.data >>> print(data) {'Comment': 'CORINA 2.61 0041 25.10.2001', 'NSC': '1'} >>> print(len(data), data.keys(), data.has_key("NSC")) 2 ['Comment', 'NSC'] True >>> print(data['Comment']) CORINA 2.61 0041 25.10.2001 >>> data['Comment'] = 'This is a new comment' >>> for k,v in data.items(): ... print(k, "–>", v) Comment –> This is a new comment NSC –> 1 >>> del data['NSC'] >>> print(len(data), data.keys(), data.has_key("NSC")) 1 ['Comment'] False

Methods

clear()	
has_key(key)	
items()	
iteritems()	
keys()	
to_dict()	
update(dictionary)	
values()	

clear()

has_key(*key*)

items()

iteritems()

keys()

to_dict()

update(*dictionary*)

values()

class oddt.toolkits.rdk.**Outputfile**(*format*, *filename*, *overwrite=False*)
Bases: object

Represent a file to which *output* is to be sent.

Required parameters:

format - see the outformats variable for a list of available output formats

filename

Optional parameters:

overwite – if the output file already exists, should it be overwritten? (default is False)

Methods: write(molecule) close()

Methods

close()	Close the Outputfile to further writing.
write(molecule)	Write a molecule to the output file.

close()
Close the Outputfile to further writing.

write(*molecule*)
Write a molecule to the output file.

Required parameters: molecule

class oddt.toolkits.rdk.**Residue**(*ParentMol*, *atom_path*)
Bases: object

Represent a RDKit residue.

Required parameter: ParentMol – Parent molecule (Mol) object path – atoms path of a residue

Attributes: atoms, idx, name.

(refer to the Open Babel library documentation for more info).

The Mol object constucted of residues' atoms can be accessed using the attribute: Residue

Attributes

atoms
idx
name

atoms

idx

name

class oddt.toolkits.rdk.**ResidueStack**(*Mol*, *paths*)
Bases: object

class oddt.toolkits.rdk.**Smarts**(*smartspattern*)
Bases: object

Initialise with a SMARTS pattern.

Methods

findall(molecule[, unique])	Find all matches of the SMARTS pattern to a particular molecule.
match(molecule)	Find all matches of the SMARTS pattern to a particular molecule.

findall(*molecule*, *unique=True*)
Find all matches of the SMARTS pattern to a particular molecule.

Required parameters: molecule

match(*molecule*)
Find all matches of the SMARTS pattern to a particular molecule.

Required parameters: molecule

oddt.toolkits.rdk.**base_feature_factory = <rdkit.Chem.rdMolChemicalFeatures.MolChemicalFeatureFactory obj**
Global feature factory based on BaseFeatures.fdef

oddt.toolkits.rdk.**descs = ['fr_C_O_noCOO', 'PEOE_VSA3', 'Chi4v', 'fr_Ar_COO', 'fr_SH', 'Chi4n', 'SMR_VSA10**
A list of supported descriptors

oddt.toolkits.rdk.**forcefields = ['mmff94', 'uff']**
A list of supported forcefields

oddt.toolkits.rdk.**fps = ['rdkit', 'layered', 'maccs', 'atompairs', 'torsions', 'morgan']**
A list of supported fingerprint types

oddt.toolkits.rdk.**informats = {'inchi': 'InChI', 'mol2': 'Tripos MOL2 file', 'sdf': 'MDL SDF file', 'smi': 'SMILES',**
A dictionary of supported input formats

oddt.toolkits.rdk.**outformats = {'inchikey': 'InChIKey', 'sdf': 'MDL SDF file', 'can': 'Canonical SMILES', 'smi': 'S**
A dictionary of supported output formats

oddt.toolkits.rdk.**readfile**(*format*, *filename*, *lazy=False*, *opt=None*, **args*, ***kwargs*)
Iterate over the molecules in a file.

Required parameters:

format - see the informats variable for a list of available input formats

filename

You can access the first molecule in a file using the next() method of the iterator:

mol = next(readfile("smi", "myfile.smi"))

You can make a list of the molecules in a file using: mols = list(readfile("smi", "myfile.smi"))

You can iterate over the molecules in a file as shown in the following code snippet: >>> atomtotal = 0 >>> for mol in readfile("sdf", "head.sdf"): ... atomtotal += len(mol.atoms) ... >>> print(atomtotal) 43

oddt.toolkits.rdk.**readstring**(*format*, *string*, ***kwargs*)
Read in a molecule from a string.

Required parameters:

format - see the informats variable for a list of available input formats

string

Example: >>> input = "C1=CC=CS1" >>> mymol = readstring("smi", input) >>> len(mymol.atoms) 5

Module contents

5.1.2 Submodules

5.1.3 oddt.datasets module

Datasets wrapped in conviniet models

class oddt.datasets.**pdbbind**(*home*, *version=None*, *default_set=None*, *data_file=None*, *opt=None*)
Bases: object

Attributes

activities
ids

activities

ids

5.1.4 oddt.fingerprints module

Module checks interactions between two molecules and creates interacion fingerprints.

oddt.fingerprints.**InteractionFingerprint**(*ligand*, *protein*, *strict=True*)

Interaction fingerprint accomplished by converting the molecular interaction of ligand-protein into bit array according to the residue of choice and the interaction. For every residue (One row = one residue) there are eight bits which represent eight type of interactions:

- (Column 0) hydrophobic contacts
- (Column 1) aromatic face to face
- (Column 2) aromatic edge to face
- (Column 3) hydrogen bond (protein as hydrogen bond donor)
- (Column 4) hydrogen bond (protein as hydrogen bond acceptor)
- (Column 5) salt bridges (protein positively charged)
- (Column 6) salt bridges (protein negatively charged)
- (Column 7) salt bridges (ionic bond with metal ion)

Parameters **ligand, protein** : oddt.toolkit.Molecule object

Molecules, which are analysed in order to find interactions.

strict : bool (deafult = True)

If False, do not include condition, which informs whether atoms form 'strict' H-bond (pass all angular cutoffs).

Returns **InteractionFingerprint** : numpy array

Vector of calculated IFP (size = no residues * 8 type of interaction)

oddt.fingerprints.**SimpleInteractionFingerprint**(*ligand*, *protein*, *strict=True*)

Based on http://dx.doi.org/10.1016/j.csbj.2014.05.004. Every IFP consists of 8 bits per amino acid (One row = one amino acid) and present eight type of interaction:

- (Column 0) hydrophobic contacts
- (Column 1) aromatic face to face
- (Column 2) aromatic edge to face
- (Column 3) hydrogen bond (protein as hydrogen bond donor)
- (Column 4) hydrogen bond (protein as hydrogen bond acceptor)
- (Column 5) salt bridges (protein positively charged)
- (Column 6) salt bridges (protein negatively charged)
- (Column 7) salt bridges (ionic bond with metal ion)

Returns matrix, which is sorted acordingly to this pattern : 'ALA', 'ARG', 'ASN', 'ASP', 'CYS', 'GLN', 'GLU', 'GLY', 'HIS', 'ILE', 'LEU', 'LYS', 'MET', 'PHE', 'PRO', 'SER', 'THR', 'TRP', 'TYR', 'VAL', ''. The '' means cofactor. Index of amino acid in pattern coresponds to row in returned matrix.

Parameters **ligand, protein** : oddt.toolkit.Molecule object

Molecules, which are analysed in order to find interactions.

strict : bool (deafult = True)

If False, do not include condition, which informs whether atoms form 'strict' H-bond (pass all angular cutoffs).

Returns **InteractionFingerprint** : numpy array

Vector of calculated IFP (size = 168)

oddt.fingerprints.**SPLIF**(*ligand*, *protein*, *depth=1*, *size=4096*, *distance_cutoff=4.5*)

Calculates structural protein-ligand interaction fingerprint (SPLIF), based on http://pubs.acs.org/doi/abs/10.1021/ci500319f.

Parameters **ligand, protein** : oddt.toolkit.Molecule object

Molecules, which are analysed in order to find interactions.

depth : int (deafult = 1)

The depth of the fingerprint, i.e. the number of bonds in Morgan algorithm. Note: For ECFP2: depth = 1, ECFP4: depth = 2, etc.

size: int (default = 4096)

SPLIF is folded to given size.

distance_cutoff: float (default=4.5)

Cutoff distance for close contacts.

Returns **SPLIF** : numpy array

Calculated SPLIF.shape = (no. of atoms,). Every row consists of three elements: row[0] = index of hashed atoms row[1].shape = (7, 3) -> ligand's atom coords and 6 his neigbor's row[2].shape = (7, 3) -> protein's atom coords and 6 his neigbor's

oddt.fingerprints.**similarity_SPLIF**(*reference*, *query*, *rmsd_cutoff=1.0*)

Calculates similarity between structural interaction fingerprints, based on doi:http://pubs.acs.org/doi/abs/10.1021/ci500319f.

Parameters **reference, query: numpy.array**

SPLIFs, which are compared in order to determine similarity.

rmsd_cutoff : int (default = 1)

Specific treshold for which, bits are considered as fully matching.

Returns **SimilarityScore** : float

Similarity between given fingerprints.

oddt.fingerprints.**ECFP**(*mol*, *depth=2*, *size=4096*, *count_bits=True*, *sparse=True*, *use_pharm_features=False*)

Extended connectivity fingerprints (ECFP) with an option to include atom features (FCPF). Depth of a fingerprint is counted as bond-steps, thus the depth for ECFP2 = 1, ECPF4 = 2, ECFP6 = 3, etc.

Reference: Rogers D, Hahn M. Extended-connectivity fingerprints. J Chem Inf Model. 2010;50: 742-754. http://dx.doi.org/10.1021/ci100050t

Parameters **mol** : oddt.toolkit.Molecule object

Input molecule for the FP calculations

depth : int (deafult = 2)

The depth of the fingerprint, i.e. the number of bonds in Morgan algorithm. Note: For ECFP2: depth = 1, ECFP4: depth = 2, etc.

size : int (default = 4096)

Final size of fingerprint to which it is folded.

count_bits : bool (default = True)

Should the bits be counted or unique. In dense representation it translates to integer array (count_bits=True) or boolean array if False.

sparse : bool (default=True)

Should fingerprints be dense (contain all bits) or sparse (just the on bits).

use_pharm_features : bool (default=False)

Switch to use pharmacophoric features as atom representation instead of explicit atomic numbers etc.

Returns **fingerprint** : numpy array

Calsulated FP of fixed size (dense) or on bits indices (sparse). Dtype is either integer or boolean.

oddt.fingerprints.**dice**(*a*, *b*, *sparse=False*)

Calculates the Dice coefficient, the ratio of the bits in common to the arithmetic mean of the number of 'on' bits in the two fingerprints. Supports integer and boolean fingerprints.

Parameters **a, b** : numpy array

Interaction fingerprints, which are compared in order to determine similarity.

sparse : bool (default=False)

Type of FPs to use. Defaults to dense form.

Returns **score** : float

Similarity between a, b.

oddt.fingerprints.**tanimoto**(*a*, *b*, *sparse=False*)

Tanimoto coefficient, supports boolean fingerprints. Integer fingerprints are casted to boolean.

Parameters **a, b** : numpy array

Interaction fingerprints, which are compared in order to determine similarity.

sparse : bool (default=False)

Type of FPs to use. Defaults to dense form.

Returns **score** : float

Similarity between a, b.

5.1.5 oddt.interactions module

Module calculates interactions between two molecules (proein-protein, protein-ligand, small-small). Currently following interacions are implemented:

- hydrogen bonds
- halogen bonds
- pi stacking (parallel and perpendicular)
- salt bridges
- hydrophobic contacts

- pi-cation
- metal coordination
- pi-metal

oddt.interactions.**close_contacts**(*x*, *y*, *cutoff*, *x_column='coords'*, *y_column='coords'*)

Returns pairs of atoms which are within close contac distance cutoff.

Parameters **x, y** : atom_dict-type numpy array

Atom dictionaries generated by oddt.toolkit.Molecule objects.

cutoff [float] Cutoff distance for close contacts

x_column, ycolumn [string, (default='coords')] Column containing coordinates of atoms (or pseudo-atoms, i.e. ring centroids)

Returns **x_, y_** : atom_dict-type numpy array

Aligned pairs of atoms in close contact for further processing.

oddt.interactions.**hbond_acceptor_donor**(*mol1*, *mol2*, *cutoff=3.5*, *base_angle=120*, *tolerance=30*)

Returns pairs of acceptor-donor atoms, which meet H-bond criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute H-bond acceptor and H-bond donor pairs

cutoff [float, (default=3.5)] Distance cutoff for A-D pairs

base_angle [int, (default=120)] Base angle determining allowed direction of hydrogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which H-bonds are considered as strict.

Returns **a, d** : atom_dict-type numpy array

Aligned arrays of atoms forming H-bond, firstly acceptors, secondly donors.

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' H-bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**hbonds**(*mol1*, *mol2*, **args*, ***kwargs*)

Calculates H-bonds between molecules

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute H-bond acceptor and H-bond donor pairs

cutoff [float, (default=3.5)] Distance cutoff for A-D pairs

base_angle [int, (default=120)] Base angle determining allowed direction of hydrogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which H-bonds are considered as strict.

Returns **mol1_atoms, mol2_atoms** : atom_dict-type numpy array

Aligned arrays of atoms forming H-bond

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' H-bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**halogenbond_acceptor_halogen**(*mol1*, *mol2*, *base_angle_acceptor=120*, *base_angle_halogen=180*, *tolerance=30*, *cutoff=4*)

Returns pairs of acceptor-halogen atoms, which meet halogen bond criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute halogen bond acceptor and halogen pairs

cutoff [float, (default=4)] Distance cutoff for A-H pairs

base_angle_acceptor [int, (default=120)] Base angle determining allowed direction of halogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

base_angle_halogen [int (default=180)] Ideal base angle between halogen bond and halogen-neighbor bond

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which halogen bonds are considered as strict.

Returns **a, h** : atom_dict-type numpy array

Aligned arrays of atoms forming halogen bond, firstly acceptors, secondly halogens

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' halogen bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**halogenbonds**(*mol1*, *mol2*, ***kwargs*)

Calculates halogen bonds between molecules

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute halogen bond acceptor and halogen pairs

cutoff [float, (default=4)] Distance cutoff for A-H pairs

base_angle_acceptor [int, (default=120)] Base angle determining allowed direction of halogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

base_angle_halogen [int (default=180)] Ideal base angle between halogen bond and halogen-neighbor bond

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which halogen bonds are considered as strict.

Returns **mol1_atoms, mol2_atoms** : atom_dict-type numpy array

Aligned arrays of atoms forming halogen bond

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' halogen bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**pi_stacking**(*mol1*, *mol2*, *cutoff=5*, *tolerance=30*)

Returns pairs of rings, which meet pi stacking criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute ring pairs

cutoff [float, (default=5)] Distance cutoff for Pi-stacking pairs

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (parallel or perpendicular) in which pi-stackings are considered as strict.

Returns **r1, r2** : ring_dict-type numpy array

Aligned arrays of rings forming pi-stacking

strict_parallel [numpy array, dtype=bool] Boolean array align with ring pairs, informing whether rings form 'strict' parallel pi-stacking. If false, only distance cutoff is met, therefore the stacking is 'crude'.

strict_perpendicular [numpy array, dtype=bool] Boolean array align with ring pairs, informing whether rings form 'strict' perpendicular pi-stacking (T-shaped, T-face, etc.). If false, only distance cutoff is met, therefore the stacking is 'crude'.

oddt.interactions.**salt_bridge_plus_minus**(*mol1*, *mol2*, *cutoff=4*)

Returns pairs of plus-mins atoms, which meet salt bridge criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute plus and minus pairs

cutoff [float, (default=4)] Distance cutoff for A-H pairs

Returns **plus, minus** : atom_dict-type numpy array

Aligned arrays of atoms forming salt bridge, firstly plus, secondly minus

oddt.interactions.**salt_bridges**(*mol1*, *mol2*, **args*, ***kwargs*)

Calculates salt bridges between molecules

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute plus and minus pairs

cutoff [float, (default=4)] Distance cutoff for plus-minus pairs

Returns **mol1_atoms, mol2_atoms** : atom_dict-type numpy array

Aligned arrays of atoms forming salt bridges

oddt.interactions.**hydrophobic_contacts**(*mol1*, *mol2*, *cutoff=4*)

Calculates hydrophobic contacts between molecules

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute hydrophobe pairs

cutoff [float, (default=4)] Distance cutoff for hydrophobe pairs

Returns **mol1_atoms, mol2_atoms** : atom_dict-type numpy array

Aligned arrays of atoms forming hydrophobic contacts

oddt.interactions.**pi_cation**(*mol1*, *mol2*, *cutoff=5*, *tolerance=30*)

Returns pairs of ring-cation atoms, which meet pi-cation criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute ring-cation pairs

cutoff [float, (default=5)] Distance cutoff for Pi-cation pairs

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (perpendicular) in which pi-cation are considered as strict.

Returns **r1** : ring_dict-type numpy array

Aligned rings forming pi-stacking

plus2 [atom_dict-type numpy array] Aligned cations forming pi-cation

strict_parallel [numpy array, dtype=bool] Boolean array align with ring-cation pairs, informing whether they form 'strict' pi-cation. If false, only distance cutoff is met, therefore the interaction is 'crude'.

oddt.interactions.**acceptor_metal**(*mol1*, *mol2*, *base_angle=120*, *tolerance=30*, *cutoff=4*)

Returns pairs of acceptor-metal atoms, which meet metal coordination criteria Note: This function is directional (mol1 holds acceptors, mol2 holds metals)

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute acceptor and metal pairs

cutoff [float, (default=4)] Distance cutoff for A-M pairs

base_angle [int, (default=120)] Base angle determining allowed direction of metal coordination, which is devided by the number of neighbors of acceptor atom to establish final directional angle

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in metal coordination are considered as strict.

Returns **a, d** : atom_dict-type numpy array

Aligned arrays of atoms forming metal coordination, firstly acceptors, secondly metals.

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' metal coordination (pass all angular cutoffs). If false, only distance cutoff is met, therefore the interaction is 'crude'.

oddt.interactions.**pi_metal**(*mol1*, *mol2*, *cutoff=5*, *tolerance=30*)

Returns pairs of ring-metal atoms, which meet pi-metal criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute ring-metal pairs

cutoff [float, (default=5)] Distance cutoff for Pi-metal pairs

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (perpendicular) in which pi-metal are considered as strict.

Returns **r1** : ring_dict-type numpy array

Aligned rings forming pi-metal

m [atom_dict-type numpy array] Aligned metals forming pi-metal

strict_parallel [numpy array, dtype=bool] Boolean array align with ring-metal pairs, informing whether they form 'strict' pi-metal. If false, only distance cutoff is met, therefore the interaction is 'crude'.

5.1.6 oddt.metrics module

Metrics for estimating performance of drug discovery methods implemented in ODDT

`oddt.metrics.roc`(*y_true*, *y_score*, *pos_label=None*, *sample_weight=None*, *drop_intermediate=True*)

Compute Receiver operating characteristic (ROC)

Note: this implementation is restricted to the binary classification task.

Read more in the User Guide.

Parameters **y_true** : array, shape = [n_samples]

True binary labels in range {0, 1} or {-1, 1}. If labels are not binary, pos_label should be explicitly given.

y_score : array, shape = [n_samples]

Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

pos_label : int or str, default=None

Label considered as positive and others are considered negative.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

drop_intermediate : boolean, optional (default=True)

Whether to drop some suboptimal thresholds which would not appear on a plotted ROC curve. This is useful in order to create lighter ROC curves.

New in version 0.17: parameter *drop_intermediate*.

Returns **fpr** : array, shape = [>2]

Increasing false positive rates such that element i is the false positive rate of predictions with score >= thresholds[i].

tpr : array, shape = [>2]

Increasing true positive rates such that element i is the true positive rate of predictions with score >= thresholds[i].

thresholds : array, shape = [n_thresholds]

Decreasing thresholds on the decision function used to compute fpr and tpr. *thresholds[0]* represents no instances being predicted and is arbitrarily set to *max(y_score) + 1*.

See also:

roc_auc_score Compute Area Under the Curve (AUC) from prediction scores

Notes

Since the thresholds are sorted from low to high values, they are reversed upon returning them to ensure they correspond to both `fpr` and `tpr`, which are sorted in reversed order during their calculation.

References

[R1]

Examples

```
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([1, 1, 2, 2])
>>> scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, scores, pos_label=2)
>>> fpr
array([ 0. ,  0.5,  0.5,  1. ])
>>> tpr
array([ 0.5,  0.5,  1. ,  1. ])
>>> thresholds
array([ 0.8 ,  0.4 ,  0.35,  0.1 ])
```

oddt.metrics.**auc**(*x*, *y*, *reorder=False*)

Compute Area Under the Curve (AUC) using the trapezoidal rule

This is a general function, given points on a curve. For computing the area under the ROC-curve, see `roc_auc_score()`.

Parameters **x** : array, shape = [n]

x coordinates.

y : array, shape = [n]

y coordinates.

reorder : boolean, optional (default=False)

If True, assume that the curve is ascending in the case of ties, as for an ROC curve. If the curve is non-ascending, the result will be wrong.

Returns **auc** : float

See also:

roc_auc_score Computes the area under the ROC curve

precision_recall_curve Compute precision-recall pairs for different probability thresholds

Examples

```
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([1, 1, 2, 2])
>>> pred = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, pred, pos_label=2)
>>> metrics.auc(fpr, tpr)
0.75
```

oddt.metrics.**roc_auc**(*y_true*, *y_score*, *pos_label=None*, *ascending_score=True*)

Computes ROC AUC score

Parameters **y_true** : array, shape=[n_samples]

True binary labels, in range {0,1} or {-1,1}. If positive label is different than 1, it must be explicitly defined.

y_score [array, shape=[n_samples]] Scores for tested series of samples

pos_label: int Positive label of samples (if other than 1)

ascending_score: bool (default=True) Indicates if your score is ascendig. Ascending score icreases with deacreasing activity. In other words it ascends on ranking list (where actives are on top).

Returns **ef** : float

Enrichment Factor for given percenage in range 0:1

oddt.metrics.**roc_log_auc**(*y_true*, *y_score*, *pos_label=None*, *ascending_score=True*, *log_min=0.001*, *log_max=1.0*)

Computes area under semi-log ROC for random distribution.

Parameters **y_true** : array, shape=[n_samples]

True binary labels, in range {0,1} or {-1,1}. If positive label is different than 1, it must be explicitly defined.

y_score [array, shape=[n_samples]] Scores for tested series of samples

pos_label: int Positive label of samples (if other than 1)

ascending_score: bool (default=True) Indicates if your score is ascendig. Ascending score icreases with deacreasing activity. In other words it ascends on ranking list (where actives are on top).

log_min [float (default=0.001)] Minimum logarithm value for estimating AUC

log_max [float (default=1.)] Maximum logarithm value for estimating AUC.

Returns **auc** : float

semi-log ROC AUC

oddt.metrics.**enrichment_factor**(*y_true*, *y_score*, *percentage=1*, *pos_label=None*, *kind='fold'*)

Computes enrichment factor for given percentage, i.e. EF_1% is enrichment factor for first percent of given samples.

Parameters **y_true** : array, shape=[n_samples]

True binary labels, in range {0,1} or {-1,1}. If positive label is different than 1, it must be explicitly defined.

y_score [array, shape=[n_samples]] Scores for tested series of samples

percentage [int or float] The percentage for which EF is being calculated

pos_label: int Positive label of samples (if other than 1)

kind: 'fold' or 'percentage' (default='fold') Two kinds of enrichment factor: fold and percentage. Fold shows the increase over random distribution (1 is random, the higher EF the better enrichment). Percentage returns the fraction of positive labels within the top x% of dataset.

Returns **ef** : float

Enrichment Factor for given percenage in range 0:1

oddt.metrics.**random_roc_log_auc**(*log_min=0.001*, *log_max=1.0*)

Computes area under semi-log ROC for random distribution.

Parameters **log_min** : float (default=0.001)

Minimum logarithm value for estimating AUC

log_max [float (default=1.)] Maximum logarithm value for estimating AUC.

Returns **auc** : float

semi-log ROC AUC for random distribution

oddt.metrics.**rmse**(*y_true*, *y_pred*)

Compute Root Mean Squared Error (RMSE)

Parameters **y_true** : array-like of shape = [n_samples] or [n_samples, n_outputs]

Ground truth (correct) target values.

y_pred [array-like of shape = [n_samples] or [n_samples, n_outputs]] Estimated target values.

Returns **rmse** : float

A positive floating point value (the best value is 0.0).

5.1.7 oddt.pandas module

Pandas extension for chemical analysis

class oddt.pandas.**ChemDataFrame**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

Bases: pandas.core.frame.DataFrame

Chemical DataFrame object, which contains molecules column of *oddt.toolkit.Molecule* objects. Rich display of moleucles (2D) is available in iPython Notebook. Additional *to_sdf* and *to_mol2* methods make writing to molecular formats easy.

New in version 0.3.

Note: Thanks to: http://blog.snapdragon.cc/2015/05/05/subclass-pandas-dataframe-to-save-custom-attributes/

Attributes

T	Transpose index and columns
at	Fast label-based scalar accessor
axes	Return a list with the row axis labels and column axis labels as the only members.
blocks	Internal property, property synonym for as_blocks()
dtypes	Return the dtypes in this object.
empty	True if NDFrame is entirely empty [no items], meaning any of the axes are of length 0.
ftypes	Return the ftypes (indication of sparse/dense and dtype) in this object.
iat	Fast integer location scalar accessor.
iloc	Purely integer-location based indexing for selection by position.
ix	A primarily label-location based indexer, with integer position fallback.
loc	Purely label-location based indexer for selection by label.
ndim	Number of axes / array dimensions
shape	Return a tuple representing the dimensionality of the DataFrame.
size	number of elements in the NDFrame
style	Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.
values	Numpy representation of NDFrame

is_copy	

Methods

abs()	Return an object with absolute value taken–only applicable to objects that are all numeric.
add(other[, axis, level, fill_value])	Addition of dataframe and other, element-wise (binary operator *add*).
add_prefix(prefix)	Concatenate prefix string with panel items names.
add_suffix(suffix)	Concatenate suffix string with panel items names.
agg(func[, axis])	Aggregate using callable, string, dict, or list of string/callables
aggregate(func[, axis])	Aggregate using callable, string, dict, or list of string/callables
align(other[, join, axis, level, copy, ...])	Align two object on their axes with the
all([axis, bool_only, skipna, level])	Return whether all elements are True over requested axis
any([axis, bool_only, skipna, level])	Return whether any element is True over requested axis
append(other[, ignore_index, verify_integrity])	Append rows of *other* to the end of this frame, returning a new object.

Continued on next page

Table 5.43 – continued from previous page

apply(func[, axis, broadcast, raw, reduce, args])	Applies function along input axis of DataFrame.
applymap(func)	Apply a function to a DataFrame that is intended to operate elementwise, i.e.
as_blocks([copy])	Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
as_matrix([columns])	Convert the frame to its Numpy-array representation.
asfreq(freq[, method, how, normalize, ...])	Convert TimeSeries to specified frequency.
asof(where[, subset])	The last row without any NaN is taken (or the last row without
assign(**kwargs)	Assign new columns to a DataFrame, returning a new object (a copy) with all the original columns in addition to the new ones.
astype(*args, **kwargs)	Cast object to input numpy.dtype
at_time(time[, asof])	Select values at particular time of day (e.g.
between_time(start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
bfill([axis, inplace, limit, downcast])	Synonym for `DataFrame.fillna(method='bfill')`
bool()	Return the bool of a single element PandasObject.
boxplot([column, by, ax, fontsize, rot, ...])	Make a box plot from DataFrame column optionally grouped by some columns or
clip([lower, upper, axis])	Trim values at input threshold(s).
clip_lower(threshold[, axis])	Return copy of the input with values below given value(s) truncated.
clip_upper(threshold[, axis])	Return copy of input with values above given value(s) truncated.
combine(other, func[, fill_value, overwrite])	Add two DataFrame objects and do not propagate NaN values, so if for a
combine_first(other)	Combine two DataFrame objects and default to non-null values in frame calling the method.
compound([axis, skipna, level])	Return the compound percentage of the values for the requested axis
consolidate([inplace])	DEPRECATED: consolidate will be an internal implementation only.
convert_objects([convert_dates, ...])	Deprecated.
copy([deep])	Make a copy of this objects data.
corr([method, min_periods])	Compute pairwise correlation of columns, excluding NA/null values
corrwith(other[, axis, drop])	Compute pairwise correlation between rows or columns of two DataFrame objects.
count([axis, level, numeric_only])	Return Series with number of non-NA/null observations over requested axis.
cov([min_periods])	Compute pairwise covariance of columns, excluding NA/null values
cummax([axis, skipna])	Return cumulative max over requested axis.
cummin([axis, skipna])	Return cumulative minimum over requested axis.
cumprod([axis, skipna])	Return cumulative product over requested axis.
cumsum([axis, skipna])	Return cumulative sum over requested axis.
describe([percentiles, include, exclude])	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.

Continued on next page

Table 5.43 – continued from previous page

diff([periods, axis])	1st discrete difference of object
div(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *truediv*).
divide(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *truediv*).
dot(other)	Matrix multiplication with DataFrame or Series objects
drop(labels[, axis, level, inplace, errors])	Return new object with labels in requested axis removed.
drop_duplicates([subset, keep, inplace])	Return DataFrame with duplicate rows removed, optionally only
dropna([axis, how, thresh, subset, inplace])	Return object with labels on given axis omitted where alternately any
duplicated([subset, keep])	Return boolean Series denoting duplicate rows, optionally only
eq(other[, axis, level])	Wrapper for flexible comparison methods eq
equals(other)	Determines if two NDFrame objects contain the same elements.
eval(expr[, inplace])	Evaluate an expression in the context of the calling DataFrame instance.
ewm([com, span, halflife, alpha, ...])	Provides exponential weighted functions
expanding([min_periods, freq, center, axis])	Provides expanding transformations.
ffill([axis, inplace, limit, downcast])	Synonym for `DataFrame.fillna(method='ffill')`
fillna([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method
filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.
first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
first_valid_index()	Return label for first non-NA/null value
floordiv(other[, axis, level, fill_value])	Integer division of dataframe and other, element-wise (binary operator *floordiv*).
from_csv(path[, header, sep, index_col, ...])	Read CSV file (DISCOURAGED, please use `pandas.read_csv()` instead).
from_dict(data[, orient, dtype])	Construct DataFrame from dict of array-like or dicts
from_items(items[, columns, orient])	Convert (key, value) pairs to DataFrame.
from_records(data[, index, exclude, ...])	Convert structured or record ndarray to DataFrame
ge(other[, axis, level])	Wrapper for flexible comparison methods ge
get(key[, default])	Get item from object for given key (DataFrame column, Panel slice, etc.).
get_dtype_counts()	Return the counts of dtypes in this object.
get_ftype_counts()	Return the counts of ftypes in this object.
get_value(index, col[, takeable])	Quickly retrieve single value at passed column and index
get_values()	same as values (but handles sparseness conversions)
groupby([by, axis, level, as_index, sort, ...])	Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.
gt(other[, axis, level])	Wrapper for flexible comparison methods gt
head([n])	Returns first n rows
hist(data[, column, by, grid, xlabelsize, ...])	Draw histogram of the DataFrame's series using matplotlib / pylab.

Continued on next page

Table 5.43 – continued from previous page

idxmax([axis, skipna])	Return index of first occurrence of maximum over requested axis.
idxmin([axis, skipna])	Return index of first occurrence of minimum over requested axis.
info([verbose, buf, max_cols, memory_usage, ...])	Concise summary of a DataFrame.
insert(loc, column, value[, allow_duplicates])	Insert column into DataFrame at specified location.
interpolate([method, axis, limit, inplace, ...])	Interpolate values according to different methods.
isin(values)	Return boolean DataFrame showing whether each element in the DataFrame is contained in values.
isnull()	Return a boolean same-sized object indicating if the values are null.
iteritems()	Iterator over (column name, Series) pairs.
iterrows()	Iterate over DataFrame rows as (index, Series) pairs.
itertuples([index, name])	Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.
join(other[, on, how, lsuffix, rsuffix, sort])	Join columns with other DataFrame either on index or on a key column.
keys()	Get the 'info axis' (see Indexing for more)
kurt([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
kurtosis([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.
last_valid_index()	Return label for last non-NA/null value
le(other[, axis, level])	Wrapper for flexible comparison methods le
lookup(row_labels, col_labels)	Label-based "fancy indexing" function for DataFrame.
lt(other[, axis, level])	Wrapper for flexible comparison methods lt
mad([axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis
mask(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.
max([axis, skipna, level, numeric_only])	This method returns the maximum of the values in the object.
mean([axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis
median([axis, skipna, level, numeric_only])	Return the median of the values for the requested axis
melt([id_vars, value_vars, var_name, ...])	"Unpivots" a DataFrame from wide format to long format, optionally
memory_usage([index, deep])	Memory usage of DataFrame columns.
merge(right[, how, on, left_on, right_on, ...])	Merge DataFrame objects by performing a database-style join operation by columns or indexes.
min([axis, skipna, level, numeric_only])	This method returns the minimum of the values in the object.
mod(other[, axis, level, fill_value])	Modulo of dataframe and other, element-wise (binary operator *mod*).
mode([axis, numeric_only])	Gets the mode(s) of each element along the axis selected.

Continued on next page

Table 5.43 – continued from previous page

mul(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *mul*).
multiply(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *mul*).
ne(other[, axis, level])	Wrapper for flexible comparison methods ne
nlargest(n, columns[, keep])	Get the rows of a DataFrame sorted by the *n* largest values of *columns*.
notnull()	Return a boolean same-sized object indicating if the values are not null.
nsmallest(n, columns[, keep])	Get the rows of a DataFrame sorted by the *n* smallest values of *columns*.
nunique([axis, dropna])	Return Series with number of distinct observations over requested axis.
pct_change([periods, fill_method, limit, freq])	Percent change over given number of periods.
pipe(func, *args, **kwargs)	Apply func(self, *args, **kwargs)
pivot([index, columns, values])	Reshape data (produce a "pivot" table) based on column values.
pivot_table(data[, values, index, columns, ...])	Create a spreadsheet-style pivot table as a DataFrame.
plot	alias of `FramePlotMethods`
pop(item)	Return item and drop from frame.
pow(other[, axis, level, fill_value])	Exponential power of dataframe and other, element-wise (binary operator *pow*).
prod([axis, skipna, level, numeric_only])	Return the product of the values for the requested axis
product([axis, skipna, level, numeric_only])	Return the product of the values for the requested axis
quantile([q, axis, numeric_only, interpolation])	Return values at the given quantile over requested axis, a la numpy.percentile.
query(expr[, inplace])	Query the columns of a frame with a boolean expression.
radd(other[, axis, level, fill_value])	Addition of dataframe and other, element-wise (binary operator *radd*).
rank([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
rdiv(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
reindex([index, columns])	Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_axis(labels[, axis, method, level, ...])	Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_like(other[, method, copy, limit, ...])	Return an object with matching indices to myself.
rename([index, columns])	Alter axes input function or functions.
rename_axis(mapper[, axis, copy, inplace])	Alter index and / or columns using input function or functions.
reorder_levels(order[, axis])	Rearrange index levels using input order.
replace([to_replace, value, inplace, limit, ...])	Replace values given in 'to_replace' with 'value'.
resample(rule[, how, axis, fill_method, ...])	Convenience method for frequency conversion and resampling of time series.
reset_index([level, drop, inplace, ...])	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.

Continued on next page

Table 5.43 – continued from previous page

rfloordiv(other[, axis, level, fill_value])	Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).
rmod(other[, axis, level, fill_value])	Modulo of dataframe and other, element-wise (binary operator *rmod*).
rmul(other[, axis, level, fill_value])	Multiplication of dataframe and other, element-wise (binary operator *rmul*).
rolling(window[, min_periods, freq, center, ...])	Provides rolling window calculcations.
round([decimals])	Round a DataFrame to a variable number of decimal places.
rpow(other[, axis, level, fill_value])	Exponential power of dataframe and other, element-wise (binary operator *rpow*).
rsub(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *rsub*).
rtruediv(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *rtruediv*).
sample([n, frac, replace, weights, ...])	Returns a random sample of items from an axis of object.
select(crit[, axis])	Return data corresponding to axis labels matching criteria
select_dtypes([include, exclude])	Return a subset of a DataFrame including/excluding columns based on their `dtype`.
sem([axis, skipna, level, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(axis, labels)	public verson of axis assignment
set_index(keys[, drop, append, inplace, ...])	Set the DataFrame index (row labels) using one or more existing columns.
set_value(index, col, value[, takeable])	Put single value at passed column and index
shift([periods, freq, axis])	Shift index by desired number of periods with an optional time freq
skew([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis
slice_shift([periods, axis])	Equivalent to *shift* without copying data.
sort_index([axis, level, ascending, ...])	Sort object by labels (along an axis)
sort_values(by[, axis, ascending, inplace, ...])	Sort by the values along either axis
sortlevel([level, axis, ascending, inplace, ...])	DEPRECATED: use `DataFrame.sort_index()`
squeeze([axis])	Squeeze length 1 dimensions.
stack([level, dropna])	Pivot a level of the (possibly hierarchical) column labels, returning a DataFrame (or Series in the case of an object with a single level of column labels) having a hierarchical index with a new inner-most level of row labels.
std([axis, skipna, level, ddof, numeric_only])	Return sample standard deviation over requested axis.
sub(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *sub*).
subtract(other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator *sub*).
sum([axis, skipna, level, numeric_only])	Return the sum of the values for the requested axis
swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately
swaplevel([i, j, axis])	Swap levels i and j in a MultiIndex on a particular axis
tail([n])	Returns last n rows
take(indices[, axis, convert, is_copy])	Analogous to ndarray.take

Continued on next page

Table 5.43 – continued from previous page

to_clipboard([excel, sep])	Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.
to_csv(*args, **kwargs)	Write DataFrame to a comma-separated values (csv) file
to_dense()	Return dense representation of NDFrame (as opposed to sparse)
to_dict([orient])	Convert DataFrame to dictionary.
to_excel(*args, **kwargs)	Write DataFrame to an excel sheet
to_feather(fname)	write out the binary feather-format for DataFrames
to_gbq(destination_table, project_id[, ...])	Write a DataFrame to a Google BigQuery table.
to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
to_html(*args, **kwargs)	Render a DataFrame as an HTML table.
to_json([path_or_buf, orient, date_format, ...])	Convert the object to a JSON string.
to_latex([buf, columns, col_space, header, ...])	Render a DataFrame to a tabular environment table.
to_mol2([filepath_or_buffer, ...])	Write DataFrame to Mol2 file.
to_msgpack([path_or_buf, encoding])	msgpack (serialize) object to input file path
to_panel()	Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.
to_period([freq, axis, copy])	Convert DataFrame from DatetimeIndex to PeriodIndex with desired
to_pickle(path[, compression])	Pickle (serialize) object to input file path.
to_records([index, convert_datetime64])	Convert DataFrame to record array.
to_sdf([filepath_or_buffer, ...])	Write DataFrame to SDF file.
to_sparse([fill_value, kind])	Convert to SparseDataFrame
to_sql(name, con[, flavor, schema, ...])	Write records stored in a DataFrame to a SQL database.
to_stata(fname[, convert_dates, ...])	A class for writing Stata binary dta files from array-like objects
to_string([buf, columns, col_space, header, ...])	Render a DataFrame to a console-friendly tabular output.
to_timestamp([freq, how, axis, copy])	Cast to DatetimeIndex of timestamps, at *beginning* of period
to_xarray()	Return an xarray object from the pandas object.
transform(func, *args, **kwargs)	Call function producing a like-indexed NDFrame
transpose(*args, **kwargs)	Transpose index and columns
truediv(other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator *truediv*).
truncate([before, after, axis, copy])	Truncates a sorted NDFrame before and/or after some particular index value.
tshift([periods, freq, axis])	Shift the time index, using the index's frequency if available.
tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
tz_localize(*args, **kwargs)	Localize tz-naive TimeSeries to target time zone.
unstack([level, fill_value])	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.
update(other[, join, overwrite, ...])	Modify DataFrame in place using non-NA values from passed DataFrame.
var([axis, skipna, level, ddof, numeric_only])	Return unbiased variance over requested axis.

Continued on next page

Table 5.43 – continued from previous page

where(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.
xs(key[, axis, level, drop_level])	Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

T
Transpose index and columns

abs()
Return an object with absolute value taken–only applicable to objects that are all numeric.

Returns abs: type of caller

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.radd`

Notes

Mismatched indices will be unioned together

add_prefix(*prefix*)
Concatenate prefix string with panel items names.

Parameters **prefix** : string

Returns **with_prefix** : type of caller

add_suffix(*suffix*)
Concatenate suffix string with panel items names.

Parameters **suffix** : string

Returns **with_suffix** : type of caller

agg(*func*, *axis=0*, **args*, ***kwargs*)
Aggregate using callable, string, dict, or list of string/callables

New in version 0.20.0.

Parameters **func** : callable, string, dictionary, or list of string/callables

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)

Returns **aggregated** : DataFrame

See also:

`pandas.DataFrame.apply`, `pandas.DataFrame.transform`, `pandas.DataFrame.groupby.aggregate`, `pandas.DataFrame.resample.aggregate`, `pandas.DataFrame.rolling.aggregate`

Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., np.mean(arr_2d, axis=0)) as opposed to mimicking the default Numpy behavior (e.g., np.mean(arr_2d)).

agg is an alias for aggregate. Use it.

Examples

```
>>> 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
```

Aggregate these functions across all columns

```
>>> df.agg(['sum', 'min'])
            A         B         C
sum -0.182253 -0.614014 -2.909534
min -1.916563 -1.460076 -1.568297
```

Different aggregations per column

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
            A         B
max       NaN  1.514318
min -1.916563 -1.460076
sum -0.182253       NaN
```

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

Aggregate using callable, string, dict, or list of string/callables

New in version 0.20.0.

Parameters **func** : callable, string, dictionary, or list of string/callables

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)

Returns **aggregated** : DataFrame

See also:

`pandas.DataFrame.apply`, `pandas.DataFrame.transform`, `pandas.DataFrame.groupby.aggregate`, `pandas.DataFrame.resample.aggregate`, `pandas.DataFrame.rolling.aggregate`

Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., np.mean(arr_2d, axis=0)) as opposed to mimicking the default Numpy behavior (e.g., np.mean(arr_2d)).

agg is an alias for aggregate. Use it.

Examples

```
>>> 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
```

Aggregate these functions across all columns

```
>>> df.agg(['sum', 'min'])
            A         B         C
sum -0.182253 -0.614014 -2.909534
min -1.916563 -1.460076 -1.568297
```

Different aggregations per column

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
            A         B
max       NaN  1.514318
min -1.916563 -1.460076
sum -0.182253       NaN
```

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 object on their axes with the specified join method for each axis Index

Parameters **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

New in version 0.17.0.

Returns (left, right) : (DataFrame, type of other)

Aligned objects

all(*axis=None*, *bool_only=None*, *skipna=None*, *level=None*, ***kwargs*)
Return whether all elements are True over requested axis

Parameters 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

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.

Returns all : Series or DataFrame (if level specified)

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

Parameters 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

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.

Returns **any** : Series or DataFrame (if level specified)

append(*other*, *ignore_index=False*, *verify_integrity=False*)

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

Parameters **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.

Returns **appended** : DataFrame

See also:

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

Notes

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.

Examples

```
>>> 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
```

apply (*func*, *axis=0*, *broadcast=False*, *raw=False*, *reduce=None*, *args=()*, ***kwds*)
Applies function along input axis of DataFrame.

Objects passed to functions are Series objects having index either the DataFrame's index (axis=0) or the columns (axis=1). Return type depends on whether passed function aggregates, or the reduce argument if the DataFrame is empty.

Parameters **func** : function

Function to apply to each column/row

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

broadcast : boolean, default False

For aggregation functions, return object of same size with values propagated

raw : boolean, default False

If False, convert each row or column into a Series. If raw=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 : boolean 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 is None (the default), apply's return value will be guessed by calling func an empty Series (note: while guessing, exceptions raised by func will be ignored). If reduce is True a Series will always be returned, and if False a DataFrame will always be returned.

args : tuple

Positional arguments to pass to function in addition to the array/series

Additional keyword arguments will be passed as keywords to the function

Returns **applied** : Series or DataFrame

See also:

DataFrame.applymap For elementwise operations

DataFrame.aggregate only perform aggregating type operations

DataFrame.transform only perform transformating type operations

Notes

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.

Examples

```
>>> df.apply(numpy.sqrt) # returns DataFrame
>>> df.apply(numpy.sum, axis=0) # equiv to df.sum(0)
>>> df.apply(numpy.sum, axis=1) # equiv to df.sum(1)
```

applymap(*func*)

Apply a function to a DataFrame that is intended to operate elementwise, i.e. like doing map(func, series) for each series in the DataFrame

Parameters **func** : function

Python function, returns a single value from a single value

Returns **applied** : DataFrame

See also:

DataFrame.apply For operations on rows/columns

Examples

```
>>> df = pd.DataFrame(np.random.randn(3, 3))
>>> df
          0         1          2
0 -0.029638  1.081563   1.280300
1  0.647747  0.831136  -1.549481
2  0.513416 -0.884417   0.195343
>>> df = df.applymap(lambda x: '%.2f' % x)
>>> df
      0     1      2
0 -0.03  1.08   1.28
1  0.65  0.83  -1.55
2  0.51 -0.88   0.20
```

as_blocks(*copy=True*)

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

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

Parameters **copy** : boolean, default True

Returns **values** : a dict of dtype -> Constructor Types

as_matrix(*columns=None*)

Convert the frame to its Numpy-array representation.

Parameters **columns: list, optional, default:None**

If None, return all columns, otherwise, returns specified columns.

Returns **values** : ndarray

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

See also:

pandas.DataFrame.values

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'.

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.

Parameters **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.

Returns **converted** : type of caller

See also:

reindex

Notes

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

Examples

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
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
```

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

Parameters **where** : date or array of dates

subset : string or list of strings, default None

if not None use these columns for NaN propagation

Returns where is scalar

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

where is Index: same shape object as input

See also:

`merge_asof`

Notes

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

assign(***kwargs*)

Assign new columns to a DataFrame, returning a new object (a copy) with all the original columns in addition to the new ones.

New in version 0.16.0.

Parameters **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.

Returns **df** : DataFrame

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

Notes

Since `kwargs` is a dictionary, the order of your arguments may not be preserved. To make things predicatable, the columns are inserted in alphabetical order, at the end of your DataFrame. Assigning multiple columns within the same `assign` is possible, but you cannot reference other columns created within the same `assign` call.

Examples

```
>>> df = 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
```

astype (**args*, ***kwargs*)

Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

Parameters **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.

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 : DEPRECATED use `errors` instead

kwargs : keyword arguments to pass on to the constructor

Returns **casted** : type of caller

at

Fast label-based scalar accessor

Similarly to `loc`, `at` provides **label** based scalar lookups. You can also set using these indexers.

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

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

Parameters **time** : datetime.time or string

Returns **values_at_time** : type of caller

axes

Return a list with the row axis labels and column axis labels as the only members. They are returned in that order.

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).

Parameters **start_time** : datetime.time or string

end_time : datetime.time or string

include_start : boolean, default True

include_end : boolean, default True

Returns **values_between_time** : type of caller

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

blocks
Internal property, property synonym for as_blocks()

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 column optionally grouped by some columns or other inputs

Parameters **data** : the pandas object holding the data

column : column name or list of names, or vector

Can be any valid input to groupby

by : string or sequence

Column in the DataFrame to group by

ax : Matplotlib axes object, optional

fontsize : int or string

rot : label rotation angle

figsize : A tuple (width, height) in inches

grid : Setting this to True will show the grid

layout : tuple (optional)

(rows, columns) for the layout of the plot

return_type : {None, 'axes', 'dict', 'both'}, default None

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, unless `return_type` is None, in which case a NumPy array of axes is returned with the same shape as `layout`. See the prose documentation for more.

kwds : other plotting keyword arguments to be passed to matplotlib boxplot

function

Returns **lines** : dict

ax : matplotlib Axes

(ax, lines): namedtuple

Notes

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.

clip (*lower=None*, *upper=None*, *axis=None*, **args*, ***kwargs*)

Trim values at input threshold(s).

Parameters **lower** : float or array_like, default None

upper : float or array_like, default None

axis : int or string axis name, optional

Align object with lower and upper along the given axis.

Returns **clipped** : Series

Examples

```
>>> df
  0         1
0  0.335232 -1.256177
1 -1.367855  0.746646
2  0.027753 -1.176076
3  0.230930 -0.679613
4  1.261967  0.570967
>>> df.clip(-1.0, 0.5)
          0         1
0  0.335232 -1.000000
1 -1.000000  0.500000
2  0.027753 -1.000000
3  0.230930 -0.679613
4  0.500000  0.500000
>>> t
0   -0.3
1   -0.2
2   -0.1
3    0.0
4    0.1
dtype: float64
>>> df.clip(t, t + 1, axis=0)
          0         1
0  0.335232 -0.300000
1 -0.200000  0.746646
2  0.027753 -0.100000
3  0.230930  0.000000
4  1.100000  0.570967
```

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

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

Parameters **threshold** : float or array_like

axis : int or string axis name, optional

Align object with threshold along the given axis.

Returns **clipped** : same type as input

See also:

clip

clip_upper (*threshold*, *axis=None*)
Return copy of input with values above given value(s) truncated.

Parameters **threshold** : float or array_like

axis : int or string axis name, optional

Align object with threshold along the given axis.

Returns **clipped** : same type as input

See also:

clip

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)

Parameters **other** : DataFrame

func : function

fill_value : scalar value

overwrite : boolean, default True

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

Returns **result** : DataFrame

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

Parameters **other** : DataFrame

Returns **combined** : DataFrame

Examples

a's values prioritized, use values from b to fill holes:

```
>>> a.combine_first(b)
```

compound (*axis=None*, *skipna=None*, *level=None*)
Return the compound percentage of the values for the requested axis

Parameters **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

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.

Returns **compounded** : Series or DataFrame (if level specified)

consolidate(*inplace=False*)

DEPRECATED: consolidate will be an internal implementation only.

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

Deprecated.

Attempt to infer better dtype for object columns

Parameters **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.

Returns **converted** : same as input object

See also:

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.

copy(*deep=True*)

Make a copy of this objects data.

Parameters **deep** : boolean or string, default True

Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices or the data are copied.

Note that when `deep=True` data is copied, 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.

Returns **copy** : type of caller

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

Compute pairwise correlation of columns, excluding NA/null values

Parameters **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

Returns **y** : DataFrame

corrwith (*other*, *axis=0*, *drop=False*)
Compute pairwise correlation between rows or columns of two DataFrame objects.

Parameters **other** : DataFrame

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

Returns **correls** : Series

count (*axis=0*, *level=None*, *numeric_only=False*)
Return Series with number of non-NA/null observations over requested axis. Works with non-floating point data as well (detects NaN and None)

Parameters **axis** : {0 or 'index', 1 or 'columns'}, default 0

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

level : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

numeric_only : boolean, default False

Include only float, int, boolean data

Returns **count** : Series (or DataFrame if level specified)

cov (*min_periods=None*)
Compute pairwise covariance of columns, excluding NA/null values

Parameters **min_periods** : int, optional

Minimum number of observations required per pair of columns to have a valid result.

Returns **y** : DataFrame

Notes

y contains the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1 (unbiased estimator).

cummax (*axis=None*, *skipna=True*, **args*, ***kwargs*)
Return cumulative max over requested axis.

Parameters **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

Returns **cummax** : Series

See also:

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

cummin(*axis=None*, *skipna=True*, **args*, ***kwargs*)
Return cumulative minimum over requested axis.

Parameters **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

Returns **cummin** : Series

See also:

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

cumprod(*axis=None*, *skipna=True*, **args*, ***kwargs*)
Return cumulative product over requested axis.

Parameters **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

Returns **cumprod** : Series

See also:

pandas.core.window.Expanding.prod Similar functionality but ignores `NaN` values.

cumsum(*axis=None*, *skipna=True*, **args*, ***kwargs*)
Return cumulative sum over requested axis.

Parameters **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

Returns **cumsum** : Series

See also:

pandas.core.window.Expanding.sum Similar functionality but ignores `NaN` values.

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.

Parameters **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 categorical objects submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`)
- 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 select numeric types submit `numpy.number`. To select categorical objects submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`)
- None (default) : The result will exclude nothing.

Returns summary: Series/DataFrame of summary statistics

See also:

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

Notes

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 `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`.

Examples

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([[1, 'a'], [2, 'b'], [3, 'c']],
...                   columns=['numeric', 'object'])
>>> 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')
        numeric object
count       3.0      3
unique      NaN      3
top         NaN      b
freq        NaN      1
mean        2.0    NaN
std         1.0    NaN
```

```
min          1.0    NaN
25%          1.5    NaN
50%          2.0    NaN
75%          2.5    NaN
max          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         b
freq        1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       object
count       3
unique      3
top         b
freq        1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
       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
```

diff (*periods=1*, *axis=0*)
1st discrete difference of object

Parameters **periods** : int, default 1

Periods to shift for forming difference

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

Take difference over rows (0) or columns (1).

Returns **diffed** : DataFrame

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.rtruediv`

Notes

Mismatched indices will be unioned together

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns result : DataFrame

See also:

`DataFrame.rtruediv`

Notes

Mismatched indices will be unioned together

dot (*other*)
Matrix multiplication with DataFrame or Series objects

Parameters other : DataFrame or Series

Returns dot_product : DataFrame or Series

drop (*labels*, *axis=0*, *level=None*, *inplace=False*, *errors='raise'*)
Return new object with labels in requested axis removed.

Parameters labels : single label or list-like

axis : int or axis name

level : int or level name, default None

For MultiIndex

inplace : bool, default False

If True, do operation inplace and return None.

errors : {'ignore', 'raise'}, default 'raise'

If 'ignore', suppress error and existing labels are dropped.

New in version 0.16.1.

Returns dropped : type of caller

drop_duplicates (*subset=None*, *keep='first'*, *inplace=False*)
Return DataFrame with duplicate rows removed, optionally only considering certain columns

Parameters 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

Returns deduplicated : DataFrame

dropna (*axis=0*, *how='any'*, *thresh=None*, *subset=None*, *inplace=False*)

Return object with labels on given axis omitted where alternately any or all of the data are missing

Parameters **axis** : {0 or 'index', 1 or 'columns'}, or tuple/list thereof

Pass tuple or list to drop on multiple axes

how : {'any', 'all'}

- any : if any NA values are present, drop that label
- all : if all values are NA, drop that label

thresh : int, default None

int value : require that many non-NA values

subset : array-like

Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include

inplace : boolean, default False

If True, do operation inplace and return None.

Returns **dropped** : DataFrame

Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0], [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5]],
...                   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
```

Drop the columns where all elements are nan:

```
>>> df.dropna(axis=1, how='all')
     A    B  D
0  NaN  2.0  0
1  3.0  4.0  1
2  NaN  NaN  5
```

Drop the columns where any of the elements is nan

```
>>> df.dropna(axis=1, how='any')
   D
0  0
1  1
2  5
```

Drop the rows where all of the elements are nan (there is no row to drop, so df stays the same):

```
>>> df.dropna(axis=0, how='all')
     A    B   C  D
0  NaN  2.0 NaN  0
```

```
1  3.0  4.0 NaN  1
2  NaN  NaN NaN  5
```

Keep only the rows with at least 2 non-na values:

```
>>> df.dropna(thresh=2)
     A    B   C  D
0  NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
```

dtypes
Return the dtypes in this object.

duplicated(*subset=None*, *keep='first'*)
Return boolean Series denoting duplicate rows, optionally only considering certain columns

Parameters **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`.

Returns **duplicated** : Series

empty
True if NDFrame is entirely empty [no items], meaning any of the axes are of length 0.

See also:

`pandas.Series.dropna`, `pandas.DataFrame.dropna`

Notes

If NDFrame contains only NaNs, it is still not considered empty. See the example below.

Examples

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
```

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=None*, ***kwargs*)
Evaluate an expression in the context of the calling DataFrame instance.

Parameters **expr** : string

The expression string to evaluate.

inplace : bool

If the expression contains an assignment, whether to return a new DataFrame or mutate the existing.

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.

New in version 0.18.0.

kwargs : dict

See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

Returns **ret** : ndarray, scalar, or pandas object

See also:

`pandas.DataFrame.query`, `pandas.DataFrame.assign`, `pandas.eval`

Notes

For more details see the API documentation for `eval()`. For detailed examples see enhancing performance with eval.

Examples

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = DataFrame(randn(10, 2), columns=list('ab'))
>>> df.eval('a + b')
>>> df.eval('c = a + b')
```

ewm(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *freq=None*, *adjust=True*, *ignore_na=False*, *axis=0*)
Provides exponential weighted functions

New in version 0.18.0.

Parameters **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).

freq : None or string alias / date offset object, default=None (DEPRECATED)

Frequency to conform to before computing statistic

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

Returns a Window sub-classed for the particular operation

Notes

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.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

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

Examples

```
>>> 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
```

expanding (*min_periods=1*, *freq=None*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

Parameters **min_periods** : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None) (DEPRECATED)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

axis : int or string, default 0

Returns a Window sub-classed for the particular operation

Notes

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`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

Examples

```
>>> 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
```

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

Parameters **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)

Returns **filled** : DataFrame

See also:

reindex, *asfreq*

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.

Parameters **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

Returns same type as input object

See also:

`pandas.DataFrame.select`

Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

Examples

```
>>> 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
```

first(*offset*)
Convenience method for subsetting initial periods of time series data based on a date offset.

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.first('10D') -> First 10 days

first_valid_index()
Return label for first non-NA/null value

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.rfloordiv`

Notes

Mismatched indices will be unioned together

from_csv(*path*, *header=0*, *sep=','*, *index_col=0*, *parse_dates=True*, *encoding=None*, *tupleize_cols=False*, *infer_datetime_format=False*)
Read CSV file (DISCOURAGED, please 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)`.

Parameters **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.

Returns **y** : DataFrame

See also:

`pandas.read_csv`

`from_dict`(*data*, *orient='columns'*, *dtype=None*)

Construct DataFrame from dict of array-like or dicts

Parameters **data** : dict

{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

Returns DataFrame

`from_items`(*items*, *columns=None*, *orient='columns'*)

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.

Parameters **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'.

Returns **frame** : DataFrame

from_records(*data*, *index=None*, *exclude=None*, *columns=None*, *coerce_float=False*, *nrows=None*)
Convert structured or record ndarray to DataFrame

Parameters **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

Returns **df** : DataFrame

ftypes
Return the ftypes (indication of sparse/dense and dtype) in this 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.

Parameters **key** : object

Returns **value** : type of items contained in object

get_dtype_counts()
Return the counts of dtypes in this object.

get_ftype_counts()
Return the counts of ftypes in this object.

get_value(*index*, *col*, *takeable=False*)
Quickly retrieve single value at passed column and index

Parameters **index** : row label

col : column label

takeable : interpret the index/col as indexers, default False

Returns **value** : scalar value

get_values()
same as values (but handles sparseness conversions)

groupby(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=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.

Parameters **by** : mapping, function, str, or iterable

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 str or list of strs may be passed to group by the columns in `self`

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

Returns GroupBy object

Examples

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

gt(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods gt

head(*n=5*)

Returns first n rows

hist(*data*, *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*)

Draw histogram of the DataFrame's series using matplotlib / pylab.

Parameters **data** : DataFrame

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

ylabelsize : int, default None

If specified changes the y-axis label size

yrot : float, default None

rotation of y axis labels

ax : matplotlib axes object, default None

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; Be aware, 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 of the figure to create in inches by default

layout : tuple, optional

Tuple of (rows, columns) for the layout of the histograms

bins : integer, default 10

Number of histogram bins to be used

kwds : other plotting keyword arguments

To be passed to hist function

iat

Fast integer location scalar accessor.

Similarly to `iloc`, `iat` provides **integer** based lookups. You can also set using these indexers.

idxmax(*axis=0*, *skipna=True*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

Parameters **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 first index.

Returns idxmax : Series

See also:

`Series.idxmax`

Notes

This method is the DataFrame version of `ndarray.argmax`.

idxmin (*axis=0*, *skipna=True*)
Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

Parameters 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

Returns idxmin : Series

See also:

`Series.idxmin`

Notes

This method is the DataFrame version of `ndarray.argmin`.

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

info (*verbose=None*, *buf=None*, *max_cols=None*, *memory_usage=None*, *null_counts=None*)
Concise summary of a DataFrame.

Parameters verbose : {None, True, False}, optional

Whether to print the full summary. None follows the *display.max_info_columns* setting. True or False overrides the *display.max_info_columns* setting.

buf : writable buffer, defaults to sys.stdout

max_cols : int, default None

Determines whether full summary or short summary is printed. None follows the *display.max_info_columns* setting.

memory_usage : boolean/string, default None

Specifies whether total memory usage of the DataFrame elements (including index) should be displayed. None follows the *display.memory_usage* setting. True or False overrides the *display.memory_usage* setting. A value of 'deep' is equivalent of True, with deep introspection. Memory usage is shown in human-readable units (base-2 representation).

null_counts : boolean, default None

Whether to show the non-null counts

- If None, then only show if the frame is smaller than max_info_rows and max_info_columns.
- If True, always show counts.
- If False, never show counts.

insert (*loc*, *column*, *value*, *allow_duplicates=False*)

Insert column into DataFrame at specified location.

If *allow_duplicates* is False, raises Exception if column is already contained in the DataFrame.

Parameters **loc** : int

Must have 0 <= loc <= len(columns)

column : object

value : scalar, Series, or array-like

interpolate (*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

Parameters **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'

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.17.0.

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.

Returns Series or DataFrame of same shape interpolated at the NaNs

See also:

reindex, *replace*, *fillna*

Examples

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 = None

isin(*values*)

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

Parameters **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.

Returns DataFrame of booleans

Examples

When `values` is a list:

```
>>> df = 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 = 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 = 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
```

isnull()

Return a boolean same-sized object indicating if the values are null.

See also:

notnull boolean inverse of isnull

iteritems()

Iterator over (column name, Series) pairs.

See also:

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.

Returns **it** : generator

A generator that iterates over the rows of the frame.

See also:

itertuples Iterate over DataFrame rows as namedtuples of the values.

iteritems Iterate over (column name, Series) pairs.

Notes

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.

`itertuples`(*index=True*, *name='Pandas'*)

Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.

Parameters **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.

See also:

iterrows Iterate over DataFrame rows as (index, Series) pairs.

iteritems Iterate over (column name, Series) pairs.

Notes

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.

Examples

```
>>> 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.

`.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 hierachical 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.

Parameters **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 : column name, tuple/list of column names, or array-like

Column(s) in the caller to join on the index in other, otherwise joins index-on-index. If multiples columns given, the passed DataFrame must have a MultiIndex. Can pass an array as the join key if 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)

Returns joined : DataFrame

See also:

`DataFrame.merge` For column(s)-on-columns(s) operations

Notes

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

Examples

```
>>> 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
    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
```

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

Parameters **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

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.

Returns **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

Parameters **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

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.

Returns **kurt** : Series or DataFrame (if level specified)

last(*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.last('5M') -> Last 5 months

last_valid_index()

Return label for last non-NA/null value

le(*other*, *axis='columns'*, *level=None*)

Wrapper for flexible comparison methods le

loc

Purely label-location based indexer for selection by label.

`.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'` (note that contrary to usual python slices, **both** the start and the stop are included!).
- 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)

`.loc` will raise a `KeyError` when the items are not found.

See more at Selection by Label

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.

Parameters **row_labels** : sequence

The row labels to use for lookup

col_labels : sequence

The column labels to use for lookup

Notes

Akin to:

```
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

Examples

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

Parameters **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

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.

Returns **mad** : Series or DataFrame (if level specified)

mask (*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *try_cast=False*, *raise_on_error=True*)
Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.

Parameters **cond** : boolean NDFrame, array-like, or callable

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

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

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)

Returns **wh** : same type as caller

See also:

`DataFrame.where()`

Notes

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
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
```

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`.

Parameters **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

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.

Returns **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

Parameters **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

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.

Returns **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

Parameters **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

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.

Returns **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.

Parameters **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.

See also:

melt, *pivot_table*, `DataFrame.pivot`

Examples

```
>>> 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*)

Memory usage of DataFrame columns.

Parameters **index** : bool

Specifies whether to include memory usage of DataFrame's index in returned Series. If *index=True* (default is False) the first index of the Series is *Index*.

deep : bool

Introspect the data deeply, interrogate *object* dtypes for system-level memory consumption

Returns **sizes** : Series

A series with column names as index and memory usage of columns with units of bytes.

See also:

`numpy.ndarray.nbytes`

Notes

Memory usage does not include memory consumed by elements that are not components of the array if deep=False

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*)

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.

Parameters **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

Field names to join on. Must be found in both DataFrames. If on is None and not merging on indexes, then it merges on the intersection of the columns by default.

left_on : label or list, or array-like

Field names to join on in left DataFrame. Can be a vector or list of vectors of the length of the DataFrame to use a particular vector as the join key instead of columns

right_on : label or list, or array-like

Field names to join on in right DataFrame or vector/list of vectors per left_on docs

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.

New in version 0.17.0.

Returns **merged** : DataFrame

The output type will the be same as 'left', if it is a subclass of DataFrame.

See also:

`merge_ordered`, `merge_asof`

Examples

```
>>> 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
```

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`.

Parameters **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

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.

Returns **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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.rmod`

Notes

Mismatched indices will be unioned together

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])`

Parameters **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

Returns **modes** : DataFrame (sorted)

Examples

```
>>> 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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.rmul`

Notes

Mismatched indices will be unioned together

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.rmul`

Notes

Mismatched indices will be unioned together

ndim
Number of axes / array dimensions

ne(*other*, *axis='columns'*, *level=None*)
Wrapper for flexible comparison methods ne

nlargest(*n*, *columns*, *keep='first'*)
Get the rows of a DataFrame sorted by the *n* largest values of *columns*.

New in version 0.17.0.

Parameters **n** : int

Number of items to retrieve

columns : list or str

Column name or names to order by

keep : {'first', 'last', False}, default 'first'

Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

Returns DataFrame

Examples

```
>>> df = DataFrame({'a': [1, 10, 8, 11, -1],
...                 'b': list('abdce'),
...                 'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nlargest(3, 'a')
    a  b   c
3  11  c   3
1  10  b   2
2   8  d NaN
```

notnull()
Return a boolean same-sized object indicating if the values are not null.

See also:

isnull boolean inverse of notnull

nsmallest(*n*, *columns*, *keep='first'*)
Get the rows of a DataFrame sorted by the *n* smallest values of *columns*.

New in version 0.17.0.

Parameters **n** : int

Number of items to retrieve

columns : list or str

Column name or names to order by

keep : {'first', 'last', False}, default 'first'

Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

Returns DataFrame

Examples

```
>>> df = 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.

Parameters **axis** : {0 or 'index', 1 or 'columns'}, default 0

dropna : boolean, default True

Don't include NaN in the counts.

Returns **nunique** : Series

Examples

```
>>> 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*)

Percent change over given number of periods.

Parameters **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())

Returns **chg** : NDFrame

Notes

By default, the percentage change is calculated along the stat axis: 0, or `Index`, for `DataFrame` and 1, or `minor` for `Panel`. You can change this with the `axis` keyword argument.

pipe (*func*, **args*, ***kwargs*)
Apply func(self, *args, **kwargs)

New in version 0.16.2.

Parameters **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 : positional arguments passed into `func`.

kwargs : a dictionary of keyword arguments passed into `func`.

Returns **object** : the return type of `func`.

See also:

`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`

Notes

Use `.pipe` when chaining together functions that expect on Series or DataFrames. 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)
...  )
```

pivot (*index=None*, *columns=None*, *values=None*)
Reshape data (produce a "pivot" table) based on column values. Uses unique values from index / columns to form axes of the resulting DataFrame.

Parameters **index** : string or object, optional

Column name to use to make new frame's index. If None, uses existing index.

columns : string or object

Column name to use to make new frame's columns

values : string or object, optional

Column name 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

Returns **pivoted** : DataFrame

See also:

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

Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods

Examples

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

```
>>> df.pivot(index='foo', columns='bar', values='baz')
     A   B   C
one  1   2   3
two  4   5   6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
     A   B   C
one  1   2   3
two  4   5   6
```

pivot_table(*data*, *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

Parameters **data** : 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 or list of functions, 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)

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.

Returns **table** : DataFrame

See also:

`DataFrame.pivot` pivot without aggregation that can handle non-numeric data

Examples

```
>>> df
   A   B   C      D
0  foo one small  1
1  foo one large  2
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
          small  large
foo  one  1      4
     two  6      NaN
bar  one  5      4
     two  6      7
```

`plot`
alias of `FramePlotMethods`

`pop`(*item*)
Return item and drop from frame. Raise KeyError if not found.

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.rpow`

Notes

Mismatched indices will be unioned together

prod(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the product of the values for the requested axis

Parameters **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

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.

Returns **prod** : Series or DataFrame (if level specified)

product(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the product of the values for the requested axis

Parameters **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

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.

Returns **prod** : Series or DataFrame (if level specified)

quantile (*q=0.5*, *axis=0*, *numeric_only=True*, *interpolation='linear'*)

Return values at the given quantile over requested axis, a la numpy.percentile.

Parameters **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

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.

Returns **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.

Examples

```
>>> df = 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
```

query (*expr*, *inplace=False*, ***kwargs*)

Query the columns of a frame with a boolean expression.

New in version 0.13.

Parameters **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()`.

Returns **q** : DataFrame

See also:

`pandas.eval`, `DataFrame.eval`

Notes

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.

For further details and examples see the `query` documentation in indexing.

Examples

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = 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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.add`

Notes

Mismatched indices will be unioned together

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

Parameters **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

Returns **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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.truediv`

Notes

Mismatched indices will be unioned together

reindex(*index=None*, *columns=None*, ***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

Parameters **index, columns** : array-like, optional (can be specified in order, or as

keywords) New labels / index to conform to. Preferably an Index object to avoid duplicating data

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`.

New in version 0.17.0.

Returns **reindexed** : DataFrame

Examples

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
IE10                  404          0.08
Chrome                200          0.02
```

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.

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

Parameters **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`.

New in version 0.17.0.

Returns **reindexed** : DataFrame

See also:

reindex, *reindex_like*

Examples

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

reindex_like (*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)
Return an object with matching indices to myself.

Parameters **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.

New in version 0.17.0.

Returns **reindexed** : same as input

Notes

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

rename(*index=None*, *columns=None*, ***kwargs*)

Alter axes input function or functions. 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. Alternatively, change `Series.name` with a scalar value (Series only).

Parameters **index, columns** : scalar, list-like, dict-like or function, optional

Scalar or list-like will alter the `Series.name` attribute, and raise on DataFrame or Panel. dict-like or functions are transformations to apply to that axis' values

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.

Returns **renamed** : DataFrame (new object)

See also:

`pandas.NDFrame.rename_axis`

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
>>> s.rename("my_name") # scalar, changes Series.name
0    1
1    2
2    3
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2)  # function, changes labels
0    1
1    2
4    3
```

```
dtype: int64
>>> s.rename({1: 3, 2: 5})  # mapping, changes labels
0    1
3    2
5    3
dtype: int64
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(2)
Traceback (most recent call last):
...
TypeError: 'int' object is not callable
>>> 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
1  2  5
2  3  6
```

rename_axis (*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter index and / or columns using input function or functions. A scalar or list-like for `mapper` will alter the `Index.name` or `MultiIndex.names` attribute. A function or dict for `mapper` will alter the labels. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is.

Parameters **mapper** : scalar, list-like, dict-like or function, optional

axis : int or string, default 0

copy : boolean, default True

Also copy underlying data

inplace : boolean, default False

Returns **renamed** : type of caller

See also:

`pandas.NDFrame.rename`, `pandas.Index.rename`

Examples

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")  # scalar, alters df.index.name
     A  B
foo
0    1  4
1    2  5
2    3  6
>>> df.rename_axis(lambda x: 2 * x)  # function: alters labels
   A  B
0  1  4
2  2  5
4  3  6
>>> df.rename_axis({"A": "ehh", "C": "see"}, axis="columns")  # mapping
```

```
   ehh  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

Parameters **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.

Returns type of caller (new object)

replace(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*, *axis=None*)

Replace values given in 'to_replace' with 'value'.

Parameters **to_replace** : str, regex, list, dict, Series, numeric, or None

- str or regex:
  - 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 and regex rules apply as above.
- dict:
  - Nested dictionaries, e.g., {'a': {'b': nan}}, are read as follows: look in column 'a' for the value 'b' and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
  - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
- 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 use to fill holes (e.g. 0), alternately a dict of values specifying 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 form 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. Otherwise, *to_replace* must be `None` because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

method : string, optional, {'pad', 'ffill', 'bfill'}

The method to use when for replacement, when `to_replace` is a `list`.

Returns filled : NDFrame

Raises 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.

ValueError

- If *to_replace* and *value* are `list` s or `ndarray` s, but they are not the same length.

See also:

`NDFrame.reindex`, `NDFrame.asfreq`, `NDFrame.fillna`

Notes

- 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.

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.

Parameters 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

label : {'right', 'left'}

Which bin edge label to label bucket with

convention : {'start', 'end', 's', 'e'}

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.

Notes

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

Examples

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 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
```

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.

Parameters **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.

Returns **resetted** : DataFrame

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.floordiv`

Notes

Mismatched indices will be unioned together

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.mod`

Notes

Mismatched indices will be unioned together

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.mul`

Notes

Mismatched indices will be unioned together

rolling(*window*, *min_periods=None*, *freq=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)
Provides rolling window calculcations.

New in version 0.18.0.

Parameters **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.

freq : string or DateOffset object, optional (default None) (DEPRECATED)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

win_type : string, default None

Provide a window type. See the notes below.

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

Returns a Window or Rolling sub-classed for the particular operation

Notes

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`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

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).

Examples

```
>>> 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
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 explicity 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
```

round (*decimals=0*, **args*, ***kwargs*)

Round a DataFrame to a variable number of decimal places.

New in version 0.17.0.

Parameters **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.

Returns DataFrame object

See also:

`numpy.around`, `Series.round`

Examples

```
>>> 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
second  0.0  1  0.58
third   0.9  0  0.49
```

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.pow`

Notes

Mismatched indices will be unioned together

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.sub`

Notes

Mismatched indices will be unioned together

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.truediv`

Notes

Mismatched indices will be unioned together

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*)
Returns a random sample of items from an axis of object.

New in version 0.16.1.

Parameters **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).

Returns A new object of same type as caller.

Examples

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
```

select (*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Parameters **crit** : function

To be called on each index (label). Should return True or False

axis : int

Returns **selection** : type of caller

select_dtypes (*include=None*, *exclude=None*)

Return a subset of a DataFrame including/excluding columns based on their `dtype`.

Parameters **include, exclude** : list-like

A list of dtypes or strings to be included/excluded. You must pass in a non-empty sequence for at least one of these.

Returns **subset** : DataFrame

The subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.

Raises 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.

TypeError

- If either of `include` or `exclude` is not a sequence

Notes

- To select all *numeric* types use the numpy dtype `numpy.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 a 'datetime64[ns, tz]' string

Examples

```
>>> df = pd.DataFrame({'a': np.random.randn(6).astype('f4'),
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df
        a      b  c
0  0.3962   True  1
1  0.1459  False  2
2  0.2623   True  1
3  0.0764  False  2
4 -0.9703   True  1
5 -1.2094  False  2
>>> df.select_dtypes(include=['float64'])
   c
0  1
1  2
2  1
3  2
4  1
5  2
>>> df.select_dtypes(exclude=['floating'])
       b
0   True
1  False
2   True
3  False
```

```
4   True
5  False
```

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

Parameters **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

degrees of freedom

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.

Returns **sem** : Series or DataFrame (if level specified)

set_axis(*axis*, *labels*)
public verson of axis assignment

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.

Parameters **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

Returns **dataframe** : DataFrame

Examples

```
>>> indexed_df = df.set_index(['A', 'B'])
>>> indexed_df2 = df.set_index(['A', [0, 1, 2, 0, 1, 2]])
>>> indexed_df3 = df.set_index([[0, 1, 2, 0, 1, 2]])
```

set_value(*index*, *col*, *value*, *takeable=False*)
Put single value at passed column and index

Parameters **index** : row label

col : column label

value : scalar value

takeable : interpret the index/col as indexers, default False

Returns **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.

shift(*periods=1*, *freq=None*, *axis=0*)
Shift index by desired number of periods with an optional time freq

Parameters **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'}

Returns **shifted** : DataFrame

Notes

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.

size
number of elements in the NDFrame

skew(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis Normalized by N-1

Parameters **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

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.

Returns **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.

Parameters **periods** : int

Number of periods to move, can be positive or negative

Returns **shifted** : same type as caller

Notes

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

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)

Parameters **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

Returns **sorted_obj** : DataFrame

sort_values(*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)
Sort by the values along either axis

New in version 0.17.0.

Parameters **by** : str or list of str

Name or list of names which refer to the axis items.

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

Axis to direct sorting

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

Returns **sorted_obj** : DataFrame

sortlevel(*level=0*, *axis=0*, *ascending=True*, *inplace=False*, *sort_remaining=True*)

DEPRECATED: use `DataFrame.sort_index()`

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)

Parameters **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.

Returns **sorted** : DataFrame

See also:

`DataFrame.sort_index`

squeeze(*axis=None*)

Squeeze length 1 dimensions.

Parameters **axis** : None, integer or string axis name, optional

The axis to squeeze if 1-sized.

New in version 0.20.0.

Returns scalar if 1-sized, else original object

stack(*level=-1*, *dropna=True*)

Pivot a level of the (possibly hierarchical) column labels, returning a DataFrame (or Series in the case of an object with a single level of column labels) having a hierarchical index with a new inner-most level of row labels. The level involved will automatically get sorted.

Parameters **level** : int, string, or list of these, default last level

Level(s) to stack, can pass level name

dropna : boolean, default True

Whether to drop rows in the resulting Frame/Series with no valid values

Returns **stacked** : DataFrame or Series

Examples

```
>>> s
     a   b
one  1.  2.
two  3.  4.
```

```
>>> s.stack()
one a    1
    b    2
two a    3
    b    4
```

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

Parameters **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

degrees of freedom

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.

Returns **std** : Series or DataFrame (if level specified)

style
Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

See also:

`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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.rsub`

Notes

Mismatched indices will be unioned together

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.rsub`

Notes

Mismatched indices will be unioned together

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the sum of the values for the requested axis

Parameters **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

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.

Returns **sum** : Series or DataFrame (if level specified)

swapaxes(*axis1*, *axis2*, *copy=True*)
Interchange axes and swap values axes appropriately

Returns **y** : same as input

swaplevel(*i=-2*, *j=-1*, *axis=0*)
Swap levels i and j in a MultiIndex on a particular axis

Parameters **i, j** : int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

Returns **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*)
Returns last n rows

take(*indices*, *axis=0*, *convert=True*, *is_copy=True*, ***kwargs*)
Analogous to ndarray.take

Parameters **indices** : list / array of ints

axis : int, default 0

convert : translate neg to pos indices (default)

is_copy : mark the returned frame as a copy

Returns **taken** : type of caller

to_clipboard(*excel=None*, *sep=None*, ***kwargs*)
Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.

Parameters **excel** : boolean, defaults to True

if True, use the provided separator, writing in a csv format for allowing easy pasting into excel. if False, write a string representation of the object to the clipboard

sep : optional, defaults to tab

other keywords are passed to to_csv

Notes

Requirements for your platform

- Linux: xclip, or xsel (with gtk or PyQt4 modules)
- Windows: none
- OS X: none

to_csv(**args*, ***kwargs*)
Write DataFrame to a comma-separated values (csv) file

Parameters **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 column names. If a list of string 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’, ‘xz’, 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

write multi_index columns as a list of tuples (if True) or new (expanded format) 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

New in version 0.16.0.

to_dense()

Return dense representation of NDFrame (as opposed to sparse)

to_dict(*orient='dict'*)

Convert DataFrame to dictionary.

Parameters 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}}

New in version 0.17.0.

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

Returns result : dict like {column -> {index -> value}}

to_excel(**args*, ***kwargs*)

Write DataFrame to an excel sheet

Parameters 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 column names. If a list of string 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.

Notes

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.

Parameters fname : str

string file path

to_gbq(*destination_table*, *project_id*, *chunksize=10000*, *verbose=True*, *reauth=False*, *if_exists='fail'*, *private_key=None*)

Write a DataFrame to a Google BigQuery table.

The main method a user calls to export pandas DataFrame contents to Google BigQuery table.

Google BigQuery API Client Library v2 for Python is used. Documentation is available here

Authentication to the Google BigQuery service is via OAuth 2.0.

- If "private_key" is not provided:

  By default "application default credentials" are used.

  If default application credentials are not found or are restrictive, user account credentials are used. In this case, you will be asked to grant permissions for product name 'pandas GBQ'.

- If "private_key" is provided:

  Service account credentials will be used to authenticate.

Parameters **dataframe** : DataFrame

DataFrame to be written

destination_table : string

Name of table to be written, in the form 'dataset.tablename'

project_id : str

Google BigQuery Account project ID.

chunksize : int (default 10000)

Number of rows to be inserted in each chunk from the dataframe.

verbose : boolean (default True)

Show percentage complete

reauth : boolean (default False)

Force Google BigQuery to reauthenticate the user. This is useful if multiple accounts are used.

if_exists : {'fail', 'replace', 'append'}, default 'fail'

'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)

to_hdf(*path_or_buf*, *key*, ***kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Parameters **path_or_buf** : the path (string) or HDFStore object

key : string

identifier for the group in the store

mode : optional, {'a', 'w', 'r+'}, default 'a'

'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+' It is similar to 'a', but the file must already exist.

format : 'fixed(f)|table(t)', default is 'fixed'

fixed(f) [Fixed format] Fast writing/reading. Not-appendable, nor searchable

table(t) [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 : boolean, default False

For Table formats, append the input data to the existing

data_columns : list of columns, or True, default None

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 here.

Applicable only to format='table'.

complevel : int, 1-9, default 0

If a complib is specified compression will be applied where possible

complib : {'zlib', 'bzip2', 'lzo', 'blosc', None}, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

fletcher32 : bool, default False

If applying compression use the fletcher32 checksum

dropna : boolean, default False.

If true, ALL nan rows will not be written to store.

to_html (**args*, ***kwargs*)

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.

Parameters **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

justify : {'left', 'right'}, default None

Left or right-justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box.

Returns **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*)

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.

Parameters **path_or_buf** : the path or buffer to write the result string

if this is None, return a StringIO of the converted string

orient : string

- 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 : The number of decimal places to use when encoding

floating point values, default 10.

force_ascii : force encoded string to be ASCII, default True.

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.

Returns same type as input object with filtered info axis

See also:

`pd.read_json`

Examples

```
>>> 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 `'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 `'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 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=True*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render a DataFrame to a tabular environment table. You can splice this into a LaTeX document. Requires usepackage{booktabs}.

to_latex-specific options:

bold_rows [boolean, default True] 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.

Parameters **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 column names. If a list of string 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

Returns **formatted** : string (or unicode, depending on data and options)

to_mol2 (*filepath_or_buffer=None*, *update_properties=True*, *molecule_column='mol'*, *columns=None*)
Write DataFrame to Mol2 file.

New in version 0.3.

Parameters **filepath_or_buffer** : string or None

File path

update_properties [bool, optional (default=True)] Switch to update properties from the DataFrames to the molecules while writting.

molecule_column [string or None, optional (default='mol')] Name of molecule column. If None the molecules will be skipped.

columns [list or None, optional (default=None)] A list of columns to write to file. If None then all available fields are written.

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.

Parameters **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.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

Returns **panel** : Panel

to_period (*freq=None*, *axis=0*, *copy=True*)
Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

Parameters **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

Returns **ts** : TimeSeries with PeriodIndex

to_pickle (*path*, *compression='infer'*)
Pickle (serialize) object to input file path.

Parameters **path** : string

File path

compression : {'infer', 'gzip', 'bz2', 'xz', None}, default 'infer'

a string representing the compression to use in the output file

New in version 0.20.0.

to_records(*index=True*, *convert_datetime64=True*)
Convert DataFrame to record array. Index will be put in the 'index' field of the record array if requested

Parameters **index** : boolean, default True

Include index in resulting record array, stored in 'index' field

convert_datetime64 : boolean, default True

Whether to convert the index to datetime.datetime if it is a DatetimeIndex

Returns **y** : recarray

to_sdf(*filepath_or_buffer=None*, *update_properties=True*, *molecule_column=None*, *columns=None*)
Write DataFrame to SDF file.

New in version 0.3.

Parameters **filepath_or_buffer** : string or None

File path

update_properties [bool, optional (default=True)] Switch to update properties from the DataFrames to the molecules while writting.

molecule_column [string or None, optional (default='mol')] Name of molecule column. If None the molecules will be skipped.

columns [list or None, optional (default=None)] A list of columns to write to file. If None then all available fields are written.

to_sparse(*fill_value=None*, *kind='block'*)
Convert to SparseDataFrame

Parameters **fill_value** : float, default NaN

kind : {'block', 'integer'}

Returns **y** : SparseDataFrame

to_sql(*name*, *con*, *flavor=None*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*)
Write records stored in a DataFrame to a SQL database.

Parameters **name** : string

Name of SQL table

con : SQLAlchemy engine or DBAPI2 connection (legacy mode)

Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

flavor : 'sqlite', default None

DEPRECATED: this parameter will be removed in a future version, as 'sqlite' is the only supported option if SQLAlchemy is not installed.

schema : string, default None

Specify the schema (if database flavor supports this). If None, use default schema.

if_exists : {'fail', 'replace', 'append'}, default 'fail'

- 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.

index : boolean, default True

Write DataFrame index as a column.

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, default None

If not None, then rows will be written in batches of this size at a time. If None, all rows will be written at once.

dtype : dict of column name to SQL type, default None

Optional specifying the datatype for columns. The SQL type should be a SQLAlchemy type, or a string for sqlite3 fallback connection.

to_stata(*fname*, *convert_dates=None*, *write_index=True*, *encoding='latin-1'*, *byteorder=None*, *time_stamp=None*, *data_label=None*, *variable_labels=None*)

A class for writing Stata binary dta files from array-like objects

Parameters **fname** : str or buffer

String path of file-like object

convert_dates : dict

Dictionary mapping columns containing datetime types to stata internal format to use when wirting 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.

dataset_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.

Raises **NotImplementedError**

- If datetimes contain timezone information
- Column dtype is not representable in Stata

ValueError

- Columns listed in convert_dates are noth either 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.

Examples

```
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

Or 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.

Parameters **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 column names. If a list of string 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

justify : {'left', 'right'}, default None

Left or right-justify the column labels. If None uses the option from the print configuration (controlled by set_option), 'right' out of the box.

Returns **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

Parameters **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

Returns **df** : DataFrame with DatetimeIndex

to_xarray()

Return an xarray object from the pandas object.

Returns a DataArray for a Series

a Dataset for a DataFrame

a DataArray for higher dims

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
```

```
   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],
        [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'
```

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.

Parameters **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)

Returns **transformed** : NDFrame

See also:

`pandas.NDFrame.aggregate`, `pandas.NDFrame.apply`

Examples

```
>>> 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
```

transpose(**args*, ***kwargs*)

Transpose index and columns

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.

Parameters **other** : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : DataFrame

See also:

`DataFrame.rtruediv`

Notes

Mismatched indices will be unioned together

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

Truncates a sorted NDFrame before and/or after some particular index value. If the axis contains only datetime values, before/after parameters are converted to datetime values.

Parameters **before** : date

Truncate before index value

after : date

Truncate after index value

axis : the truncation axis, defaults to the stat axis

copy : boolean, default is True,

return a copy of the truncated section

Returns **truncated** : type of caller

tshift(*periods=1*, *freq=None*, *axis=0*)

Shift the time index, using the index's frequency if available.

Parameters **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

Returns **shifted** : NDFrame

Notes

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

tz_convert (*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

Parameters **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

Raises **TypeError**

If the axis is tz-naive.

tz_localize (**args*, ***kwargs*)
Localize tz-naive TimeSeries to target time zone.

Parameters **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

infer_dst : boolean, default False (DEPRECATED)

Attempt to infer fall dst-transition hours based on order

Raises **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.

Parameters **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

Returns **unstacked** : DataFrame or Series

See also:

`DataFrame.pivot` Pivot a table based on column values.

`DataFrame.stack` Pivot a level of the column labels (inverse operation from *unstack*).

Examples

```
>>> 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
```

`update` (*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)

Modify DataFrame in place using non-NA values from passed DataFrame. Aligns on indices

Parameters **other** : DataFrame, or object coercible into a DataFrame

join : {'left'}, default 'left'

overwrite : boolean, default True

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

filter_func : callable(1d-array) -> 1d-array<boolean>, default None

Can choose to replace values other than NA. Return True for values that should be updated

raise_conflict : boolean

If True, will raise an error if the DataFrame and other both contain data in the same place.

values
Numpy representation of NDFrame

Notes

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 flot64 dtype.

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

Parameters **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

degrees of freedom

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.

Returns **var** : Series or DataFrame (if level specified)

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *try_cast=False*, *raise_on_error=True*)
Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

Parameters **cond** : boolean NDFrame, array-like, or callable

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

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

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)

Returns **wh** : same type as caller

See also:

`DataFrame.mask()`

Notes

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
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
```

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).

Parameters **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.

Returns **xs** : Series or DataFrame

Notes

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

Examples

```
>>> 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
second
three   5  3  5  3
```

class oddt.pandas.**ChemPanel**(*data=None*, *items=None*, *major_axis=None*, *minor_axis=None*, *copy=False*, *dtype=None*)

Bases: pandas.core.panel.Panel

Modified *pandas.Panel* to adopt higher dimension data than *ChemDataFrame*. Main purpose is to store molecular fingerprints in one column and keep 2D numpy array underneath.

New in version 0.3.

Attributes

at	Fast label-based scalar accessor
axes	Return index label(s) of the internal NDFrame
blocks	Internal property, property synonym for as_blocks()
dtypes	Return the dtypes in this object.
empty	True if NDFrame is entirely empty [no items], meaning any of the axes are of length 0.
ftypes	Return the ftypes (indication of sparse/dense and dtype) in this object.
iat	Fast integer location scalar accessor.
iloc	Purely integer-location based indexing for selection by position.
ix	A primarily label-location based indexer, with integer position fallback.
loc	Purely label-location based indexer for selection by label.
ndim	Number of axes / array dimensions
shape	Return a tuple of axis dimensions
size	number of elements in the NDFrame
values	Numpy representation of NDFrame

is_copy	

Methods

abs()	Return an object with absolute value taken–only applicable to objects that are all numeric.
add(other[, axis])	Addition of series and other, element-wise (binary operator *add*).
add_prefix(prefix)	Concatenate prefix string with panel items names.
add_suffix(suffix)	Concatenate suffix string with panel items names.
agg(func, *args, **kwargs)	
aggregate(func, *args, **kwargs)	
align(other, **kwargs)	
all([axis, bool_only, skipna, level])	Return whether all elements are True over requested axis
any([axis, bool_only, skipna, level])	Return whether any element is True over requested axis
apply(func[, axis])	Applies function along axis (or axes) of the Panel
as_blocks([copy])	Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
as_matrix()	
asfreq(freq[, method, how, normalize, ...])	Convert TimeSeries to specified frequency.
asof(where[, subset])	The last row without any NaN is taken (or the last row without
astype(*args, **kwargs)	Cast object to input numpy.dtype
at_time(time[, asof])	Select values at particular time of day (e.g.
between_time(start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
bfill([axis, inplace, limit, downcast])	Synonym for `DataFrame.fillna(method='bfill')`
bool()	Return the bool of a single element PandasObject.
clip([lower, upper, axis])	Trim values at input threshold(s).
clip_lower(threshold[, axis])	Return copy of the input with values below given value(s) truncated.
clip_upper(threshold[, axis])	Return copy of input with values above given value(s) truncated.
compound([axis, skipna, level])	Return the compound percentage of the values for the requested axis
conform(frame[, axis])	Conform input DataFrame to align with chosen axis pair.
consolidate([inplace])	DEPRECATED: consolidate will be an internal implementation only.
convert_objects([convert_dates, ...])	Deprecated.
copy([deep])	Make a copy of this objects data.
count([axis])	Return number of observations over requested axis.
cummax([axis, skipna])	Return cumulative max over requested axis.
cummin([axis, skipna])	Return cumulative minimum over requested axis.
cumprod([axis, skipna])	Return cumulative product over requested axis.
cumsum([axis, skipna])	Return cumulative sum over requested axis.
describe([percentiles, include, exclude])	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding `NaN` values.
div(other[, axis])	Floating division of series and other, element-wise (binary operator *truediv*).

Continued on next page

Table 5.45 – continued from previous page

divide(other[, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
drop(labels[, axis, level, inplace, errors])	Return new object with labels in requested axis removed.
dropna([axis, how, inplace])	Drop 2D from panel, holding passed axis constant
eq(other[, axis])	Wrapper for comparison method eq
equals(other)	Determines if two NDFrame objects contain the same elements.
ffill([axis, inplace, limit, downcast])	Synonym for `DataFrame.fillna(method='ffill')`
fillna([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method
filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.
first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
floordiv(other[, axis])	Integer division of series and other, element-wise (binary operator *floordiv*).
fromDict(data[, intersect, orient, dtype])	Construct Panel from dict of DataFrame objects
from_dict(data[, intersect, orient, dtype])	Construct Panel from dict of DataFrame objects
ge(other[, axis])	Wrapper for comparison method ge
get(key[, default])	Get item from object for given key (DataFrame column, Panel slice, etc.).
get_dtype_counts()	Return the counts of dtypes in this object.
get_ftype_counts()	Return the counts of ftypes in this object.
get_value(*args, **kwargs)	Quickly retrieve single value at (item, major, minor) location
get_values()	same as values (but handles sparseness conversions)
groupby(function[, axis])	Group data on given axis, returning GroupBy object
gt(other[, axis])	Wrapper for comparison method gt
head([n])	
interpolate([method, axis, limit, inplace, ...])	Interpolate values according to different methods.
isnull()	Return a boolean same-sized object indicating if the values are null.
iteritems()	Iterate over (label, values) on info axis
join(other[, how, lsuffix, rsuffix])	Join items with other Panel either on major and minor axes column
keys()	Get the 'info axis' (see Indexing for more)
kurt([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
kurtosis([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.
le(other[, axis])	Wrapper for comparison method le
lt(other[, axis])	Wrapper for comparison method lt
mad([axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis
major_xs(key)	Return slice of panel along major axis

Continued on next page

Table 5.45 – continued from previous page

mask(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.
max([axis, skipna, level, numeric_only])	This method returns the maximum of the values in the object.
mean([axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis
median([axis, skipna, level, numeric_only])	Return the median of the values for the requested axis
min([axis, skipna, level, numeric_only])	This method returns the minimum of the values in the object.
minor_xs(key)	Return slice of panel along minor axis
mod(other[, axis])	Modulo of series and other, element-wise (binary operator *mod*).
mul(other[, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
multiply(other[, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
ne(other[, axis])	Wrapper for comparison method ne
notnull()	Return a boolean same-sized object indicating if the values are not null.
pct_change([periods, fill_method, limit, freq])	Percent change over given number of periods.
pipe(func, *args, **kwargs)	Apply func(self, *args, **kwargs)
pop(item)	Return item and drop from frame.
pow(other[, axis])	Exponential power of series and other, element-wise (binary operator *pow*).
prod([axis, skipna, level, numeric_only])	Return the product of the values for the requested axis
product([axis, skipna, level, numeric_only])	Return the product of the values for the requested axis
radd(other[, axis])	Addition of series and other, element-wise (binary operator *radd*).
rank([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.
rdiv(other[, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
reindex([items, major_axis, minor_axis])	Conform Panel to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_axis(labels[, axis, method, level, ...])	Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_like(other[, method, copy, limit, ...])	Return an object with matching indices to myself.
rename([items, major_axis, minor_axis])	Alter axes input function or functions.
rename_axis(mapper[, axis, copy, inplace])	Alter index and / or columns using input function or functions.
replace([to_replace, value, inplace, limit, ...])	Replace values given in 'to_replace' with 'value'.
resample(rule[, how, axis, fill_method, ...])	Convenience method for frequency conversion and resampling of time series.
rfloordiv(other[, axis])	Integer division of series and other, element-wise (binary operator *rfloordiv*).
rmod(other[, axis])	Modulo of series and other, element-wise (binary operator *rmod*).
rmul(other[, axis])	Multiplication of series and other, element-wise (binary operator *rmul*).

Continued on next page

Table 5.45 – continued from previous page

round([decimals])	Round each value in Panel to a specified number of decimal places.
rpow(other[, axis])	Exponential power of series and other, element-wise (binary operator *rpow*).
rsub(other[, axis])	Subtraction of series and other, element-wise (binary operator *rsub*).
rtruediv(other[, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
sample([n, frac, replace, weights, ...])	Returns a random sample of items from an axis of object.
select(crit[, axis])	Return data corresponding to axis labels matching criteria
sem([axis, skipna, level, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(axis, labels)	public verson of axis assignment
set_value(*args, **kwargs)	Quickly set single value at (item, major, minor) location
shift([periods, freq, axis])	Shift index by desired number of periods with an optional time freq.
skew([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis
slice_shift([periods, axis])	Equivalent to *shift* without copying data.
sort_index([axis, level, ascending, ...])	Sort object by labels (along an axis)
sort_values(by[, axis, ascending, inplace, ...])	
squeeze([axis])	Squeeze length 1 dimensions.
std([axis, skipna, level, ddof, numeric_only])	Return sample standard deviation over requested axis.
sub(other[, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
subtract(other[, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
sum([axis, skipna, level, numeric_only])	Return the sum of the values for the requested axis
swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately
swaplevel([i, j, axis])	Swap levels i and j in a MultiIndex on a particular axis
tail([n])	
take(indices[, axis, convert, is_copy])	Analogous to ndarray.take
toLong(*args, **kwargs)	
to_clipboard([excel, sep])	Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.
to_dense()	Return dense representation of NDFrame (as opposed to sparse)
to_excel(path[, na_rep, engine])	Write each DataFrame in Panel to a separate excel sheet
to_frame([filter_observations])	Transform wide format into long (stacked) format as DataFrame whose columns are the Panel's items and whose index is a MultiIndex formed of the Panel's major and minor axes.
to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
to_json([path_or_buf, orient, date_format, ...])	Convert the object to a JSON string.
to_long(*args, **kwargs)	
to_msgpack([path_or_buf, encoding])	msgpack (serialize) object to input file path
to_pickle(path[, compression])	Pickle (serialize) object to input file path.

Continued on next page

Table 5.45 – continued from previous page

to_sparse(*args, **kwargs)	NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.
to_sql(name, con[, flavor, schema, ...])	Write records stored in a DataFrame to a SQL database.
to_xarray()	Return an xarray object from the pandas object.
transpose(*args, **kwargs)	Permute the dimensions of the Panel
truediv(other[, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
truncate([before, after, axis, copy])	Truncates a sorted NDFrame before and/or after some particular index value.
tshift([periods, freq, axis])	
tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
tz_localize(*args, **kwargs)	Localize tz-naive TimeSeries to target time zone.
update(other[, join, overwrite, ...])	Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel.
var([axis, skipna, level, ddof, numeric_only])	Return unbiased variance over requested axis.
where(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.
xs(key[, axis])	Return slice of panel along selected axis

abs()

Return an object with absolute value taken–only applicable to objects that are all numeric.

Returns abs: type of caller

add(*other*, *axis=0*)

Addition of series and other, element-wise (binary operator *add*). Equivalent to `panel + other`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.radd`

add_prefix(*prefix*)

Concatenate prefix string with panel items names.

Parameters **prefix** : string

Returns **with_prefix** : type of caller

add_suffix(*suffix*)

Concatenate suffix string with panel items names.

Parameters **suffix** : string

Returns **with_suffix** : type of caller

agg(*func*, **args*, ***kwargs*)

aggregate(*func*, **args*, ***kwargs*)

align(*other*, ***kwargs*)

all (*axis=None*, *bool_only=None*, *skipna=None*, *level=None*, ***kwargs*)
Return whether all elements are True over requested axis

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **all** : DataFrame or Panel (if level specified)

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

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **any** : DataFrame or Panel (if level specified)

apply (*func*, *axis='major'*, ***kwargs*)
Applies function along axis (or axes) of the Panel

Parameters **func** : function

Function to apply to each combination of 'other' axes e.g. if axis = 'items', the combination of major_axis/minor_axis will each be passed as a Series; if axis = ('items', 'major'), DataFrames of items & major axis will be passed

axis : {'items', 'minor', 'major'}, or {0, 1, 2}, or a tuple with two

axes

Additional keyword arguments will be passed as keywords to the function

Returns **result** : Panel, DataFrame, or Series

Examples

Returns a Panel with the square root of each element

```
>>> p = pd.Panel(np.random.rand(4,3,2))
>>> p.apply(np.sqrt)
```

Equivalent to p.sum(1), returning a DataFrame

```
>>> p.apply(lambda x: x.sum(), axis=1)
```

Equivalent to previous:

```
>>> p.apply(lambda x: x.sum(), axis='minor')
```

Return the shapes of each DataFrame over axis 2 (i.e the shapes of items x major), as a Series

```
>>> p.apply(lambda x: x.shape, axis=(0,1))
```

as_blocks(*copy=True*)

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

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

Parameters **copy** : boolean, default True

Returns **values** : a dict of dtype -> Constructor Types

as_matrix()

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.

Parameters **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.

Returns **converted** : type of caller

See also:

reindex

Notes

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

Examples

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
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
```

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

Parameters **where** : date or array of dates

subset : string or list of strings, default None

if not None use these columns for NaN propagation

Returns where is scalar

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

where is Index: same shape object as input

See also:

`merge_asof`

Notes

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

astype(**args*, ***kwargs*)

Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

Parameters **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.

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 : DEPRECATED use `errors` instead

kwargs : keyword arguments to pass on to the constructor

Returns **casted** : type of caller

at

Fast label-based scalar accessor

Similarly to `loc`, `at` provides **label** based scalar lookups. You can also set using these indexers.

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

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

Parameters **time** : datetime.time or string

Returns **values_at_time** : type of caller

axes

Return index label(s) of the internal NDFrame

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).

Parameters **start_time** : datetime.time or string

end_time : datetime.time or string

include_start : boolean, default True

include_end : boolean, default True

Returns **values_between_time** : type of caller

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

blocks
Internal property, property synonym for as_blocks()

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

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

Parameters **lower** : float or array_like, default None

upper : float or array_like, default None

axis : int or string axis name, optional

Align object with lower and upper along the given axis.

Returns **clipped** : Series

Examples

```
>>> df
  0         1
0  0.335232 -1.256177
1 -1.367855  0.746646
2  0.027753 -1.176076
3  0.230930 -0.679613
4  1.261967  0.570967
>>> df.clip(-1.0, 0.5)
          0         1
0  0.335232 -1.000000
1 -1.000000  0.500000
2  0.027753 -1.000000
3  0.230930 -0.679613
4  0.500000  0.500000
>>> t
0   -0.3
1   -0.2
2   -0.1
3    0.0
4    0.1
dtype: float64
>>> df.clip(t, t + 1, axis=0)
```

```
          0         1
0  0.335232 -0.300000
1 -0.200000  0.746646
2  0.027753 -0.100000
3  0.230930  0.000000
4  1.100000  0.570967
```

clip_lower(*threshold*, *axis=None*)
Return copy of the input with values below given value(s) truncated.

Parameters **threshold** : float or array_like

axis : int or string axis name, optional

Align object with threshold along the given axis.

Returns **clipped** : same type as input

See also:

clip

clip_upper(*threshold*, *axis=None*)
Return copy of input with values above given value(s) truncated.

Parameters **threshold** : float or array_like

axis : int or string axis name, optional

Align object with threshold along the given axis.

Returns **clipped** : same type as input

See also:

clip

compound(*axis=None*, *skipna=None*, *level=None*)
Return the compound percentage of the values for the requested axis

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **compounded** : DataFrame or Panel (if level specified)

conform(*frame*, *axis='items'*)
Conform input DataFrame to align with chosen axis pair.

Parameters **frame** : DataFrame

axis : {'items', 'major', 'minor'}

Axis the input corresponds to. E.g., if axis='major', then the frame's columns would be items, and the index would be values of the minor axis

Returns DataFrame

consolidate(*inplace=False*)
DEPRECATED: consolidate will be an internal implementation only.

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

Attempt to infer better dtype for object columns

Parameters **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.

Returns **converted** : same as input object

See also:

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.

copy(*deep=True*)
Make a copy of this objects data.

Parameters **deep** : boolean or string, default True

Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices or the data are copied.

Note that when `deep=True` data is copied, 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.

Returns **copy** : type of caller

count(*axis='major'*)
Return number of observations over requested axis.

Parameters **axis** : {'items', 'major', 'minor'} or {0, 1, 2}

Returns **count** : DataFrame

cummax(*axis=None*, *skipna=True*, **args*, ***kwargs*)
Return cumulative max over requested axis.

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cummax** : DataFrame

See also:

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

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

Return cumulative minimum over requested axis.

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cummin** : DataFrame

See also:

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

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

Return cumulative product over requested axis.

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cumprod** : DataFrame

See also:

pandas.core.window.Expanding.prod Similar functionality but ignores `NaN` values.

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

Return cumulative sum over requested axis.

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cumsum** : DataFrame

See also:

pandas.core.window.Expanding.sum Similar functionality but ignores `NaN` values.

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.

Parameters **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 categorical objects submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`)
- 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 select numeric types submit `numpy.number`. To select categorical objects submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`)
- None (default) : The result will exclude nothing.

Returns summary: Series/DataFrame of summary statistics

See also:

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

Notes

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 `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`.

Examples

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([[1, 'a'], [2, 'b'], [3, 'c']],
...                   columns=['numeric', 'object'])
>>> 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')
        numeric object
count       3.0      3
unique      NaN      3
top         NaN      b
freq        NaN      1
mean        2.0    NaN
std         1.0    NaN
min         1.0    NaN
25%         1.5    NaN
50%         2.0    NaN
75%         2.5    NaN
max         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         b
freq        1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       object
count       3
unique      3
top         b
freq        1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
       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
```

div(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.rtruediv`

divide(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.rtruediv`

drop(*labels*, *axis=0*, *level=None*, *inplace=False*, *errors='raise'*)

Return new object with labels in requested axis removed.

Parameters **labels** : single label or list-like

axis : int or axis name

level : int or level name, default None

For MultiIndex

inplace : bool, default False

If True, do operation inplace and return None.

errors : {'ignore', 'raise'}, default 'raise'

If 'ignore', suppress error and existing labels are dropped.

New in version 0.16.1.

Returns **dropped** : type of caller

dropna (*axis=0*, *how='any'*, *inplace=False*)
Drop 2D from panel, holding passed axis constant

Parameters **axis** : int, default 0

Axis to hold constant. E.g. axis=1 will drop major_axis entries having a certain amount of NA data

how : {'all', 'any'}, default 'any'

'any': one or more values are NA in the DataFrame along the axis. For 'all' they all must be.

inplace : bool, default False

If True, do operation inplace and return None.

Returns **dropped** : Panel

dtypes
Return the dtypes in this object.

empty
True if NDFrame is entirely empty [no items], meaning any of the axes are of length 0.

See also:

`pandas.Series.dropna`, `pandas.DataFrame.dropna`

Notes

If NDFrame contains only NaNs, it is still not considered empty. See the example below.

Examples

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
```

eq (*other*, *axis=None*)
Wrapper for comparison method eq

equals(*other*)
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

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

Parameters **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, 1, 2, 'items', 'major_axis', 'minor_axis'}

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)

Returns **filled** : Panel

See also:

reindex, *asfreq*

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.

Parameters **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

Returns same type as input object

See also:

`pandas.DataFrame.select`

Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

Examples

```
>>> 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
```

first(*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.first('10D') -> First 10 days

floordiv(*other*, *axis=0*)
Integer division of series and other, element-wise (binary operator *floordiv*). Equivalent to `panel // other`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.rfloordiv`

fromDict(*data*, *intersect=False*, *orient='items'*, *dtype=None*)
Construct Panel from dict of DataFrame objects

Parameters **data** : dict

{field : DataFrame}

intersect : boolean

Intersect indexes of input DataFrames

orient : {'items', 'minor'}, default 'items'

The "orientation" of the data. If the keys of the passed dict should be the items of the result panel, pass 'items' (default). Otherwise if the columns of the values of the passed DataFrame objects should be the items (which in the case of mixed-dtype data you should do), instead pass 'minor'

dtype : dtype, default None

Data type to force, otherwise infer

Returns Panel

from_dict(*data*, *intersect=False*, *orient='items'*, *dtype=None*)
Construct Panel from dict of DataFrame objects

Parameters **data** : dict

{field : DataFrame}

intersect : boolean

Intersect indexes of input DataFrames

orient : {'items', 'minor'}, default 'items'

The "orientation" of the data. If the keys of the passed dict should be the items of the result panel, pass 'items' (default). Otherwise if the columns of the values of the passed DataFrame objects should be the items (which in the case of mixed-dtype data you should do), instead pass 'minor'

dtype : dtype, default None

Data type to force, otherwise infer

Returns Panel

ftypes
Return the ftypes (indication of sparse/dense and dtype) in this object.

ge(*other*, *axis=None*)
Wrapper for comparison method ge

get(*key*, *default=None*)
Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

Parameters **key** : object

Returns **value** : type of items contained in object

get_dtype_counts()
Return the counts of dtypes in this object.

get_ftype_counts()
Return the counts of ftypes in this object.

get_value(**args*, ***kwargs*)
Quickly retrieve single value at (item, major, minor) location

Parameters **item** : item label (panel item)

major : major axis label (panel item row)

minor : minor axis label (panel item column)

takeable : interpret the passed labels as indexers, default False

Returns **value** : scalar value

get_values()
same as values (but handles sparseness conversions)

groupby(*function*, *axis='major'*)
Group data on given axis, returning GroupBy object

Parameters **function** : callable

Mapping function for chosen access

axis : {'major', 'minor', 'items'}, default 'major'

Returns **grouped** : PanelGroupBy

gt(*other*, *axis=None*)
Wrapper for comparison method gt

head(*n=5*)

iat
Fast integer location scalar accessor.

Similarly to `iloc`, `iat` provides **integer** based lookups. You can also set using these indexers.

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

interpolate(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *downcast=None*, ***kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

Parameters **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'

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.17.0.

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.

Returns Series or DataFrame of same shape interpolated at the NaNs

See also:

reindex, *replace*, *fillna*

Examples

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 = None

isnull()

Return a boolean same-sized object indicating if the values are null.

See also:

notnull boolean inverse of isnull

iteritems()

Iterate over (label, values) on info axis

This is index for Series, columns for DataFrame, major_axis for Panel, and so on.

ix

A primarily label-location based indexer, with integer position fallback.

`.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 hierachical 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*, *how='left'*, *lsuffix=''*, *rsuffix=''*)

Join items with other Panel either on major and minor axes column

Parameters **other** : Panel or list of Panels

Index should be similar to one of the columns in this one

how : {'left', 'right', 'outer', 'inner'}

How to handle indexes of the two objects. Default: 'left' for joining on index, None otherwise * left: use calling frame's index * right: use input frame's index * outer: form union of indexes * inner: use intersection of indexes

lsuffix : string

Suffix to use from left frame's overlapping columns

rsuffix : string

Suffix to use from right frame's overlapping columns

Returns **joined** : Panel

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

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **kurt** : DataFrame or Panel (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

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **kurt** : DataFrame or Panel (if level specified)

last(*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.last('5M') -> Last 5 months

le(*other*, *axis=None*)
Wrapper for comparison method le

loc
Purely label-location based indexer for selection by label.

`.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'` (note that contrary to usual python slices, **both** the start and the stop are included!).
- 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)

`.loc` will raise a `KeyError` when the items are not found.

See more at Selection by Label

lt(*other*, *axis=None*)
Wrapper for comparison method lt

mad(*axis=None*, *skipna=None*, *level=None*)
Return the mean absolute deviation of the values for the requested axis

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **mad** : DataFrame or Panel (if level specified)

major_xs(*key*)
Return slice of panel along major axis

Parameters **key** : object

Major axis label

Returns **y** : DataFrame

index -> minor axis, columns -> items

Notes

major_xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of major_xs functionality, see MultiIndex Slicers

mask(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *try_cast=False*, *raise_on_error=True*)
Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.

Parameters **cond** : boolean NDFrame, array-like, or callable

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

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

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)

Returns **wh** : same type as caller

See also:

`DataFrame.where()`

Notes

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
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
```

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`.

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **max** : DataFrame or Panel (if level specified)

mean (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **mean** : DataFrame or Panel (if level specified)

median(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **median** : DataFrame or Panel (if level specified)

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`.

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **min** : DataFrame or Panel (if level specified)

minor_xs(*key*)

Return slice of panel along minor axis

Parameters **key** : object

Minor axis label

Returns **y** : DataFrame

index -> major axis, columns -> items

Notes

minor_xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of minor_xs functionality, see MultiIndex Slicers

mod(*other*, *axis=0*)

Modulo of series and other, element-wise (binary operator *mod*). Equivalent to `panel % other`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.rmod`

mul(*other*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *mul*). Equivalent to `panel * other`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.rmul`

multiply(*other*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *mul*). Equivalent to `panel * other`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.rmul`

ndim

Number of axes / array dimensions

ne(*other*, *axis=None*)

Wrapper for comparison method ne

notnull()
Return a boolean same-sized object indicating if the values are not null.

See also:

isnull boolean inverse of notnull

pct_change(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)
Percent change over given number of periods.

Parameters **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())

Returns **chg** : NDFrame

Notes

By default, the percentage change is calculated along the stat axis: 0, or `Index`, for `DataFrame` and 1, or `minor` for `Panel`. You can change this with the `axis` keyword argument.

pipe(*func*, **args*, ***kwargs*)
Apply func(self, *args, **kwargs)

New in version 0.16.2.

Parameters **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 : positional arguments passed into `func`.

kwargs : a dictionary of keyword arguments passed into `func`.

Returns **object** : the return type of `func`.

See also:

`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`

Notes

Use `.pipe` when chaining together functions that expect on Series or DataFrames. 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)
...  )
```

pop (*item*)

Return item and drop from frame. Raise KeyError if not found.

pow (*other*, *axis=0*)

Exponential power of series and other, element-wise (binary operator *pow*). Equivalent to `panel ** other`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.rpow`

prod (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the product of the values for the requested axis

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **prod** : DataFrame or Panel (if level specified)

product (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the product of the values for the requested axis

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns prod : DataFrame or Panel (if level specified)

radd(*other*, *axis=0*)

Addition of series and other, element-wise (binary operator *radd*). Equivalent to `other + panel`.

Parameters other : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.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

Parameters 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

Returns ranks : same type as caller

rdiv(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *rtruediv*). Equivalent to `other / panel`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.truediv`

reindex(*items=None*, *major_axis=None*, *minor_axis=None*, ***kwargs*)

Conform Panel 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

Parameters **items, major_axis, minor_axis** : array-like, optional (can be specified in order, or as

keywords) New labels / index to conform to. Preferably an Index object to avoid duplicating data

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`.

New in version 0.17.0.

Returns **reindexed** : Panel

Examples

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
IE10                  404          0.08
Chrome                200          0.02
```

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.

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

Parameters **labels** : array-like

New labels / index to conform to. Preferably an Index object to avoid duplicating data

axis : {0, 1, 2, 'items', 'major_axis', 'minor_axis'}

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`.

New in version 0.17.0.

Returns **reindexed** : Panel

See also:

reindex, *reindex_like*

Examples

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

reindex_like(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)

Return an object with matching indices to myself.

Parameters **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.

New in version 0.17.0.

Returns **reindexed** : same as input

Notes

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

rename (*items=None*, *major_axis=None*, *minor_axis=None*, ***kwargs*)

Alter axes input function or functions. 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. Alternatively, change `Series.name` with a scalar value (Series only).

Parameters **items, major_axis, minor_axis** : scalar, list-like, dict-like or function, optional

Scalar or list-like will alter the `Series.name` attribute, and raise on DataFrame or Panel. dict-like or functions are transformations to apply to that axis' values

copy : boolean, default True

Also copy underlying data

inplace : boolean, default False

Whether to return a new Panel. 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.

Returns **renamed** : Panel (new object)

See also:

`pandas.NDFrame.rename_axis`

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
>>> s.rename("my_name") # scalar, changes Series.name
0    1
1    2
2    3
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2)  # function, changes labels
0    1
1    2
4    3
dtype: int64
>>> s.rename({1: 3, 2: 5})  # mapping, changes labels
0    1
3    2
5    3
dtype: int64
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(2)
Traceback (most recent call last):
...
TypeError: 'int' object is not callable
```

```
>>> 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
1  2  5
2  3  6
```

rename_axis(*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter index and / or columns using input function or functions. A scalar or list-like for `mapper` will alter the `Index.name` or `MultiIndex.names` attribute. A function or dict for `mapper` will alter the labels. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is.

Parameters **mapper** : scalar, list-like, dict-like or function, optional

axis : int or string, default 0

copy : boolean, default True

Also copy underlying data

inplace : boolean, default False

Returns **renamed** : type of caller

See also:

`pandas.NDFrame.rename`, `pandas.Index.rename`

Examples

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")  # scalar, alters df.index.name
     A  B
foo
0    1  4
1    2  5
2    3  6
>>> df.rename_axis(lambda x: 2 * x)  # function: alters labels
   A  B
0  1  4
2  2  5
4  3  6
>>> df.rename_axis({"A": "ehh", "C": "see"}, axis="columns")  # mapping
   ehh  B
0    1  4
1    2  5
2    3  6
```

replace(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*, *axis=None*)

Replace values given in 'to_replace' with 'value'.

Parameters **to_replace** : str, regex, list, dict, Series, numeric, or None

- str or regex:

  - 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 and regex rules apply as above.
- dict:
  - Nested dictionaries, e.g., {'a': {'b': nan}}, are read as follows: look in column 'a' for the value 'b' and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
  - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
- 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 use to fill holes (e.g. 0), alternately a dict of values specifying 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 form 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. Otherwise, *to_replace* must be `None` because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

method : string, optional, {'pad', 'ffill', 'bfill'}

The method to use when for replacement, when `to_replace` is a `list`.

Returns filled : NDFrame

Raises 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.

ValueError

- If *to_replace* and *value* are `list` s or `ndarray` s, but they are not the same length.

See also:

`NDFrame.reindex`, `NDFrame.asfreq`, `NDFrame.fillna`

Notes

- 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.

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.

Parameters **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

label : {'right', 'left'}

Which bin edge label to label bucket with

convention : {'start', 'end', 's', 'e'}

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.

Notes

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

Examples

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 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
```

rfloordiv(*other*, *axis=0*)

Integer division of series and other, element-wise (binary operator *rfloordiv*). Equivalent to `other // panel`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.floordiv`

rmod(*other*, *axis=0*)

Modulo of series and other, element-wise (binary operator *rmod*). Equivalent to `other % panel`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.mod`

rmul(*other*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *rmul*). Equivalent to `other * panel`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.mul`

round(*decimals=0*, **args*, ***kwargs*)

Round each value in Panel to a specified number of decimal places.

New in version 0.18.0.

Parameters **decimals** : int

Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

Returns Panel object

See also:

`numpy.around`

rpow (*other*, *axis=0*)

Exponential power of series and other, element-wise (binary operator *rpow*). Equivalent to `other ** panel`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.pow`

rsub (*other*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *rsub*). Equivalent to `other - panel`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.sub`

rtruediv (*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *rtruediv*). Equivalent to `other / panel`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.truediv`

sample (*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*)

Returns a random sample of items from an axis of object.

New in version 0.16.1.

Parameters **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).

Returns A new object of same type as caller.

Examples

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
```

select(*crit*, *axis=0*)
Return data corresponding to axis labels matching criteria

Parameters **crit** : function

To be called on each index (label). Should return True or False

axis : int

Returns **selection** : type of caller

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

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

ddof : int, default 1

degrees of freedom

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.

Returns **sem** : DataFrame or Panel (if level specified)

set_axis(*axis*, *labels*)
public verson of axis assignment

set_value(**args*, ***kwargs*)
Quickly set single value at (item, major, minor) location

Parameters **item** : item label (panel item)

major : major axis label (panel item row)

minor : minor axis label (panel item column)

value : scalar

takeable : interpret the passed labels as indexers, default False

Returns **panel** : Panel

If label combo is contained, will be reference to calling Panel, otherwise a new object

shape
Return a tuple of axis dimensions

shift (*periods=1*, *freq=None*, *axis='major'*)
Shift index by desired number of periods with an optional time freq. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original. This is different from the behavior of DataFrame.shift()

Parameters **periods** : int

Number of periods to move, can be positive or negative

freq : DateOffset, timedelta, or time rule string, optional

axis : {'items', 'major', 'minor'} or {0, 1, 2}

Returns **shifted** : Panel

size
number of elements in the NDFrame

skew (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis Normalized by N-1

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **skew** : DataFrame or Panel (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.

Parameters **periods** : int

Number of periods to move, can be positive or negative

Returns **shifted** : same type as caller

Notes

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

sort_index (*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*)
Sort object by labels (along an axis)

Parameters **axis** : axes 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

Returns **sorted_obj** : NDFrame

sort_values(*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)

squeeze(*axis=None*)

Squeeze length 1 dimensions.

Parameters **axis** : None, integer or string axis name, optional

The axis to squeeze if 1-sized.

New in version 0.20.0.

Returns scalar if 1-sized, else original object

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

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

ddof : int, default 1

degrees of freedom

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.

Returns **std** : DataFrame or Panel (if level specified)

sub(*other*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *sub*). Equivalent to `panel - other`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.rsub`

subtract(*other*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *sub*). Equivalent to `panel - other`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.rsub`

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the sum of the values for the requested axis

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

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.

Returns **sum** : DataFrame or Panel (if level specified)

swapaxes(*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately

Returns **y** : same as input

swaplevel(*i=-2*, *j=-1*, *axis=0*)

Swap levels i and j in a MultiIndex on a particular axis

Parameters **i, j** : int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

Returns **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*)

take(*indices*, *axis=0*, *convert=True*, *is_copy=True*, ***kwargs*)
Analogous to ndarray.take

Parameters **indices** : list / array of ints

axis : int, default 0

convert : translate neg to pos indices (default)

is_copy : mark the returned frame as a copy

Returns **taken** : type of caller

toLong(**args*, ***kwargs*)

to_clipboard(*excel=None*, *sep=None*, ***kwargs*)
Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.

Parameters **excel** : boolean, defaults to True

if True, use the provided separator, writing in a csv format for allowing easy pasting into excel. if False, write a string representation of the object to the clipboard

sep : optional, defaults to tab

other keywords are passed to to_csv

Notes

Requirements for your platform

- Linux: xclip, or xsel (with gtk or PyQt4 modules)
- Windows: none
- OS X: none

to_dense()
Return dense representation of NDFrame (as opposed to sparse)

to_excel(*path*, *na_rep=''*, *engine=None*, ***kwargs*)
Write each DataFrame in Panel to a separate excel sheet

Parameters **path** : string or ExcelWriter object

File path or existing ExcelWriter

na_rep : string, default ''

Missing data representation

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`.

Other Parameters **float_format** : string, default None

Format string for floating point numbers

cols : sequence, optional

Columns to write

header : boolean or list of string, default True

Write out column names. If a list of string 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

Notes

Keyword arguments (and na_rep) are passed to the `to_excel` method for each DataFrame written.

to_frame (*filter_observations=True*)

Transform wide format into long (stacked) format as DataFrame whose columns are the Panel's items and whose index is a MultiIndex formed of the Panel's major and minor axes.

Parameters **filter_observations** : boolean, default True

Drop (major, minor) pairs without a complete set of observations across all the items

Returns **y** : DataFrame

to_hdf (*path_or_buf*, *key*, ***kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Parameters **path_or_buf** : the path (string) or HDFStore object

key : string

identifier for the group in the store

mode : optional, {'a', 'w', 'r+'}, default 'a'

'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+' It is similar to `'a'`, but the file must already exist.

format : 'fixed(f)|table(t)', default is 'fixed'

fixed(f) [Fixed format] Fast writing/reading. Not-appendable, nor searchable

table(t) [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 : boolean, default False

For Table formats, append the input data to the existing

data_columns : list of columns, or True, default None

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 here.

Applicable only to format='table'.

complevel : int, 1-9, default 0

If a complib is specified compression will be applied where possible

complib : {'zlib', 'bzip2', 'lzo', 'blosc', None}, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

fletcher32 : bool, default False

If applying compression use the fletcher32 checksum

dropna : boolean, default False.

If true, ALL nan rows will not be written to store.

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*)

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.

Parameters **path_or_buf** : the path or buffer to write the result string

if this is None, return a StringIO of the converted string

orient : string

- 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 : The number of decimal places to use when encoding

floating point values, default 10.

force_ascii : force encoded string to be ASCII, default True.

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.

Returns same type as input object with filtered info axis

See also:

`pd.read_json`

Examples

```
>>> 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 `'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 `'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 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_long(**args*, ***kwargs*)

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.

Parameters **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_pickle(*path*, *compression='infer'*)
Pickle (serialize) object to input file path.

Parameters **path** : string

File path

compression : {'infer', 'gzip', 'bz2', 'xz', None}, default 'infer'

a string representing the compression to use in the output file

New in version 0.20.0.

to_sparse(**args*, ***kwargs*)
NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.

Convert to SparsePanel

to_sql(*name*, *con*, *flavor=None*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*)
Write records stored in a DataFrame to a SQL database.

Parameters **name** : string

Name of SQL table

con : SQLAlchemy engine or DBAPI2 connection (legacy mode)

Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

flavor : 'sqlite', default None

DEPRECATED: this parameter will be removed in a future version, as 'sqlite' is the only supported option if SQLAlchemy is not installed.

schema : string, default None

Specify the schema (if database flavor supports this). If None, use default schema.

if_exists : {'fail', 'replace', 'append'}, default 'fail'

- 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.

index : boolean, default True

Write DataFrame index as a column.

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, default None

If not None, then rows will be written in batches of this size at a time. If None, all rows will be written at once.

dtype : dict of column name to SQL type, default None

Optional specifying the datatype for columns. The SQL type should be a SQLAlchemy type, or a string for sqlite3 fallback connection.

to_xarray()

Return an xarray object from the pandas object.

Returns a DataArray for a Series

a Dataset for a DataFrame

a DataArray for higher dims

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
   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],
        [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'
```

transpose(**args*, ***kwargs*)

Permute the dimensions of the Panel

Parameters **args** : three positional arguments: each oneof

{0, 1, 2, 'items', 'major_axis', 'minor_axis'}

copy [boolean, default False] Make a copy of the underlying data. Mixed-dtype data will always result in a copy

Returns **y** : same as input

Examples

```
>>> p.transpose(2, 0, 1)
>>> p.transpose(2, 0, 1, copy=True)
```

truediv(*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

Parameters **other** : DataFrame or Panel

axis : {items, major_axis, minor_axis}

Axis to broadcast over

Returns Panel

See also:

`Panel.rtruediv`

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

Truncates a sorted NDFrame before and/or after some particular index value. If the axis contains only datetime values, before/after parameters are converted to datetime values.

Parameters **before** : date

Truncate before index value

after : date

Truncate after index value

axis : the truncation axis, defaults to the stat axis

copy : boolean, default is True,

return a copy of the truncated section

Returns **truncated** : type of caller

tshift(*periods=1*, *freq=None*, *axis='major'*)

tz_convert(*tz*, *axis=0*, *level=None*, *copy=True*)

Convert tz-aware axis to target time zone.

Parameters **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

Raises **TypeError**

If the axis is tz-naive.

tz_localize(**args*, ***kwargs*)
Localize tz-naive TimeSeries to target time zone.

Parameters **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

infer_dst : boolean, default False (DEPRECATED)

Attempt to infer fall dst-transition hours based on order

Raises **TypeError**

If the TimeSeries is tz-aware and tz is not None.

update(*other*, *join='left'*, *overwrite=True*, *filter_func=None*, *raise_conflict=False*)
Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel. Aligns on items

Parameters **other** : Panel, or object coercible to Panel

join : How to join individual DataFrames

{'left', 'right', 'outer', 'inner'}, default 'left'

overwrite : boolean, default True

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

filter_func : callable(1d-array) -> 1d-array<boolean>, default None

Can choose to replace values other than NA. Return True for values that should be updated

raise_conflict : bool

If True, will raise an error if a DataFrame and other both contain data in the same place.

values
Numpy representation of NDFrame

Notes

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 flot64 dtype.

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

Parameters **axis** : {items (0), major_axis (1), minor_axis (2)}

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 DataFrame

ddof : int, default 1

degrees of freedom

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.

Returns **var** : DataFrame or Panel (if level specified)

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *try_cast=False*, *raise_on_error=True*)
Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

Parameters **cond** : boolean NDFrame, array-like, or callable

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

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

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)

Returns **wh** : same type as caller

See also:

`DataFrame.mask()`

Notes

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
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
```

xs (*key*, *axis=1*)

Return slice of panel along selected axis

Parameters **key** : object

Label

axis : {'items', 'major', 'minor}, default 1/'major'

Returns **y** : ndim(self)-1

Notes

xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of xs functionality, see MultiIndex Slicers

class oddt.pandas.**ChemSeries**(*data=None*, *index=None*, *dtype=None*, *name=None*, *copy=False*, *fastpath=False*)

Bases: pandas.core.series.Series

Pandas Series modified to adapt *oddt.toolkit.Molecule* objects and apply molecular methods easily.

New in version 0.3.

Attributes

T	return the transpose, which is by definition self
asobject	return object Series which contains boxed values
at	Fast label-based scalar accessor
axes	Return a list of the row axis labels
base	return the base object if the memory of the underlying data is
blocks	Internal property, property synonym for as_blocks()
data	return the data pointer of the underlying data
dtype	return the dtype object of the underlying data
dtypes	return the dtype object of the underlying data
empty	
flags	return the ndarray.flags for the underlying data
ftype	return if the data is sparse|dense
ftypes	return if the data is sparse|dense
iat	Fast integer location scalar accessor.
iloc	Purely integer-location based indexing for selection by position.
imag	
is_monotonic	Return boolean if values in the object are
is_monotonic_decreasing	Return boolean if values in the object are
is_monotonic_increasing	Return boolean if values in the object are
is_unique	Return boolean if values in the object are unique
itemsize	return the size of the dtype of the item of the underlying data
ix	A primarily label-location based indexer, with integer position fallback.
loc	Purely label-location based indexer for selection by label.
name	
nbytes	return the number of bytes in the underlying data
ndim	return the number of dimensions of the underlying data,

Continued on next page

Table 5.46 – continued from previous page

real	
shape	return a tuple of the shape of the underlying data
size	return the number of elements in the underlying data
strides	return the strides of the underlying data
values	Return Series as ndarray or ndarray-like

hasnans	
is_copy	

Methods

abs()	Return an object with absolute value taken–only applicable to objects that are all numeric.
add(other[, level, fill_value, axis])	Addition of series and other, element-wise (binary operator *add*).
add_prefix(prefix)	Concatenate prefix string with panel items names.
add_suffix(suffix)	Concatenate suffix string with panel items names.
agg(func[, axis])	Aggregate using callable, string, dict, or list of string/callables
aggregate(func[, axis])	Aggregate using callable, string, dict, or list of string/callables
align(other[, join, axis, level, copy, ...])	Align two object on their axes with the
all([axis, bool_only, skipna, level])	Return whether all elements are True over requested axis
any([axis, bool_only, skipna, level])	Return whether any element is True over requested axis
append(to_append[, ignore_index, ...])	Concatenate two or more Series.
apply(func[, convert_dtype, args])	Invoke function on values of Series.
argmax([axis, skipna])	Index of first occurrence of maximum of values.
argmin([axis, skipna])	Index of first occurrence of minimum of values.
argsort([axis, kind, order])	Overrides ndarray.argsort.
as_blocks([copy])	Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
as_matrix([columns])	Convert the frame to its Numpy-array representation.
asfreq(freq[, method, how, normalize, ...])	Convert TimeSeries to specified frequency.
asof(where[, subset])	The last row without any NaN is taken (or the last row without
astype(*args, **kwargs)	Cast object to input numpy.dtype
at_time(time[, asof])	Select values at particular time of day (e.g.
autocorr([lag])	Lag-N autocorrelation
between(left, right[, inclusive])	Return boolean Series equivalent to left <= series <= right.
between_time(start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
bfill([axis, inplace, limit, downcast])	Synonym for `DataFrame.fillna(method='bfill')`
bool()	Return the bool of a single element PandasObject.
calcfp(*args, **kwargs)	Helper function to map FP calculation throuugh the series

Continued on next page

Table 5.47 – continued from previous page

cat	alias of CategoricalAccessor
clip([lower, upper, axis])	Trim values at input threshold(s).
clip_lower(threshold[, axis])	Return copy of the input with values below given value(s) truncated.
clip_upper(threshold[, axis])	Return copy of input with values above given value(s) truncated.
combine(other, func[, fill_value])	Perform elementwise binary operation on two Series using given function
combine_first(other)	Combine Series values, choosing the calling Series's values first.
compound([axis, skipna, level])	Return the compound percentage of the values for the requested axis
compress(condition, *args, **kwargs)	Return selected slices of an array along given axis as a Series
consolidate([inplace])	DEPRECATED: consolidate will be an internal implementation only.
convert_objects([convert_dates, ...])	Deprecated.
copy([deep])	Make a copy of this objects data.
corr(other[, method, min_periods])	Compute correlation with *other* Series, excluding missing values
count([level])	Return number of non-NA/null observations in the Series
cov(other[, min_periods])	Compute covariance with Series, excluding missing values
cummax([axis, skipna])	Return cumulative max over requested axis.
cummin([axis, skipna])	Return cumulative minimum over requested axis.
cumprod([axis, skipna])	Return cumulative product over requested axis.
cumsum([axis, skipna])	Return cumulative sum over requested axis.
describe([percentiles, include, exclude])	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
diff([periods])	1st discrete difference of object
div(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
divide(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
dot(other)	Matrix multiplication with DataFrame or inner-product with Series
drop(labels[, axis, level, inplace, errors])	Return new object with labels in requested axis removed.
drop_duplicates([keep, inplace])	Return Series with duplicate values removed
dropna([axis, inplace])	Return Series without null values
dt	alias of CombinedDatetimelikeProperties
duplicated([keep])	Return boolean Series denoting duplicate values
eq(other[, level, fill_value, axis])	Equal to of series and other, element-wise (binary operator *eq*).
equals(other)	Determines if two NDFrame objects contain the same elements.
ewm([com, span, halflife, alpha, ...])	Provides exponential weighted functions
expanding([min_periods, freq, center, axis])	Provides expanding transformations.

Continued on next page

Table 5.47 – continued from previous page

factorize([sort, na_sentinel])	Encode the object as an enumerated type or categorical variable
ffill([axis, inplace, limit, downcast])	Synonym for DataFrame.fillna(method='ffill')
fillna([value, method, axis, inplace, ...])	Fill NA/NaN values using the specified method
filter([items, like, regex, axis])	Subset rows or columns of dataframe according to labels in the specified index.
first(offset)	Convenience method for subsetting initial periods of time series data based on a date offset.
first_valid_index()	Return label for first non-NA/null value
floordiv(other[, level, fill_value, axis])	Integer division of series and other, element-wise (binary operator *floordiv*).
from_array(arr[, index, name, dtype, copy, ...])	
from_csv(path[, sep, parse_dates, header, ...])	Read CSV file (DISCOURAGED, please use pandas.read_csv() instead).
ge(other[, level, fill_value, axis])	Greater than or equal to of series and other, element-wise (binary operator *ge*).
get(key[, default])	Get item from object for given key (DataFrame column, Panel slice, etc.).
get_dtype_counts()	Return the counts of dtypes in this object.
get_ftype_counts()	Return the counts of ftypes in this object.
get_value(label[, takeable])	Quickly retrieve single value at passed index label
get_values()	same as values (but handles sparseness conversions); is a view
groupby([by, axis, level, as_index, sort, ...])	Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.
gt(other[, level, fill_value, axis])	Greater than of series and other, element-wise (binary operator *gt*).
head([n])	Returns first n rows
hist([by, ax, grid, xlabelsize, xrot, ...])	Draw histogram of the input series using matplotlib
idxmax([axis, skipna])	Index of first occurrence of maximum of values.
idxmin([axis, skipna])	Index of first occurrence of minimum of values.
interpolate([method, axis, limit, inplace, ...])	Interpolate values according to different methods.
isin(values)	Return a boolean Series showing whether each element in the Series is exactly contained in the passed sequence of values.
isnull()	Return a boolean same-sized object indicating if the values are null.
item()	return the first element of the underlying data as a python
iteritems()	Lazily iterate over (index, value) tuples
keys()	Alias for index
kurt([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
kurtosis([axis, skipna, level, numeric_only])	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
last(offset)	Convenience method for subsetting final periods of time series data based on a date offset.

Continued on next page

Table 5.47 – continued from previous page

last_valid_index()	Return label for last non-NA/null value
le(other[, level, fill_value, axis])	Less than or equal to of series and other, element-wise (binary operator *le*).
lt(other[, level, fill_value, axis])	Less than of series and other, element-wise (binary operator *lt*).
mad([axis, skipna, level])	Return the mean absolute deviation of the values for the requested axis
map(arg[, na_action])	Map values of Series using input correspondence (which can be
mask(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.
max([axis, skipna, level, numeric_only])	This method returns the maximum of the values in the object.
mean([axis, skipna, level, numeric_only])	Return the mean of the values for the requested axis
median([axis, skipna, level, numeric_only])	Return the median of the values for the requested axis
memory_usage([index, deep])	Memory usage of the Series
min([axis, skipna, level, numeric_only])	This method returns the minimum of the values in the object.
mod(other[, level, fill_value, axis])	Modulo of series and other, element-wise (binary operator *mod*).
mode()	Return the mode(s) of the dataset.
mul(other[, level, fill_value, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
multiply(other[, level, fill_value, axis])	Multiplication of series and other, element-wise (binary operator *mul*).
ne(other[, level, fill_value, axis])	Not equal to of series and other, element-wise (binary operator *ne*).
nlargest([n, keep])	Return the largest *n* elements.
nonzero()	Return the indices of the elements that are non-zero
notnull()	Return a boolean same-sized object indicating if the values are not null.
nsmallest([n, keep])	Return the smallest *n* elements.
nunique([dropna])	Return number of unique elements in the object.
pct_change([periods, fill_method, limit, freq])	Percent change over given number of periods.
pipe(func, *args, **kwargs)	Apply func(self, *args, **kwargs)
plot	alias of SeriesPlotMethods
pop(item)	Return item and drop from frame.
pow(other[, level, fill_value, axis])	Exponential power of series and other, element-wise (binary operator *pow*).
prod([axis, skipna, level, numeric_only])	Return the product of the values for the requested axis
product([axis, skipna, level, numeric_only])	Return the product of the values for the requested axis
ptp([axis, skipna, level, numeric_only])	Returns the difference between the maximum value and the minimum value in the object.
put(*args, **kwargs)	Applies the *put* method to its *values* attribute if it has one.
quantile([q, interpolation])	Return value at the given quantile, a la numpy.percentile.
radd(other[, level, fill_value, axis])	Addition of series and other, element-wise (binary operator *radd*).
rank([axis, method, numeric_only, ...])	Compute numerical data ranks (1 through n) along axis.

Continued on next page

Table 5.47 – continued from previous page

ravel([order])	Return the flattened underlying data as an ndarray
rdiv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
reindex([index])	Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
reindex_axis(labels[, axis])	for compatibility with higher dims
reindex_like(other[, method, copy, limit, ...])	Return an object with matching indices to myself.
rename([index])	Alter axes input function or functions.
rename_axis(mapper[, axis, copy, inplace])	Alter index and / or columns using input function or functions.
reorder_levels(order)	Rearrange index levels using input order.
repeat(*args, **kwargs)	Repeat elements of an Series.
replace([to_replace, value, inplace, limit, ...])	Replace values given in 'to_replace' with 'value'.
resample(rule[, how, axis, fill_method, ...])	Convenience method for frequency conversion and resampling of time series.
reset_index([level, drop, name, inplace])	Analogous to the `pandas.DataFrame.reset_index()` function, see docstring there.
reshape(*args, **kwargs)	DEPRECATED: calling this method will raise an error in a future release.
rfloordiv(other[, level, fill_value, axis])	Integer division of series and other, element-wise (binary operator *rfloordiv*).
rmod(other[, level, fill_value, axis])	Modulo of series and other, element-wise (binary operator *rmod*).
rmul(other[, level, fill_value, axis])	Multiplication of series and other, element-wise (binary operator *rmul*).
rolling(window[, min_periods, freq, center, ...])	Provides rolling window calculcations.
round([decimals])	Round each value in a Series to the given number of decimals.
rpow(other[, level, fill_value, axis])	Exponential power of series and other, element-wise (binary operator *rpow*).
rsub(other[, level, fill_value, axis])	Subtraction of series and other, element-wise (binary operator *rsub*).
rtruediv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *rtruediv*).
sample([n, frac, replace, weights, ...])	Returns a random sample of items from an axis of object.
searchsorted(*args, **kwargs)	Find indices where elements should be inserted to maintain order.
select(crit[, axis])	Return data corresponding to axis labels matching criteria
sem([axis, skipna, level, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
set_axis(axis, labels)	public verson of axis assignment
set_value(label, value[, takeable])	Quickly set single value at passed label.
shift([periods, freq, axis])	Shift index by desired number of periods with an optional time freq
skew([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis
slice_shift([periods, axis])	Equivalent to *shift* without copying data.
sort_index([axis, level, ascending, ...])	Sort object by labels (along an axis)
sort_values([axis, ascending, inplace, ...])	Sort by the values along either axis

Continued on next page

Table 5.47 – continued from previous page

sortlevel([level, ascending, sort_remaining])	DEPRECATED: use Series.sort_index()
squeeze([axis])	Squeeze length 1 dimensions.
std([axis, skipna, level, ddof, numeric_only])	Return sample standard deviation over requested axis.
str	alias of StringMethods
sub(other[, level, fill_value, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
subtract(other[, level, fill_value, axis])	Subtraction of series and other, element-wise (binary operator *sub*).
sum([axis, skipna, level, numeric_only])	Return the sum of the values for the requested axis
swapaxes(axis1, axis2[, copy])	Interchange axes and swap values axes appropriately
swaplevel([i, j, copy])	Swap levels i and j in a MultiIndex
tail([n])	Returns last n rows
take(indices[, axis, convert, is_copy])	return Series corresponding to requested indices
to_clipboard([excel, sep])	Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.
to_csv([path, index, sep, na_rep, ...])	Write Series to a comma-separated values (csv) file
to_dense()	Return dense representation of NDFrame (as opposed to sparse)
to_dict()	Convert Series to {label -> value} dict
to_excel(excel_writer[, sheet_name, na_rep, ...])	Write Series to an excel sheet
to_frame([name])	Convert Series to DataFrame
to_hdf(path_or_buf, key, **kwargs)	Write the contained data to an HDF5 file using HDFStore.
to_json([path_or_buf, orient, date_format, ...])	Convert the object to a JSON string.
to_mol2([filepath_or_buffer])	
to_msgpack([path_or_buf, encoding])	msgpack (serialize) object to input file path
to_period([freq, copy])	Convert Series from DatetimeIndex to PeriodIndex with desired
to_pickle(path[, compression])	Pickle (serialize) object to input file path.
to_sdf([filepath_or_buffer])	
to_smiles([filepath_or_buffer])	
to_sparse([kind, fill_value])	Convert Series to SparseSeries
to_sql(name, con[, flavor, schema, ...])	Write records stored in a DataFrame to a SQL database.
to_string([buf, na_rep, float_format, ...])	Render a string representation of the Series
to_timestamp([freq, how, copy])	Cast to datetimeindex of timestamps, at *beginning* of period
to_xarray()	Return an xarray object from the pandas object.
tolist()	Convert Series to a nested list
transform(func, *args, **kwargs)	Call function producing a like-indexed NDFrame
transpose(*args, **kwargs)	return the transpose, which is by definition self
truediv(other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator *truediv*).
truncate([before, after, axis, copy])	Truncates a sorted NDFrame before and/or after some particular index value.
tshift([periods, freq, axis])	Shift the time index, using the index's frequency if available.
tz_convert(tz[, axis, level, copy])	Convert tz-aware axis to target time zone.
tz_localize(*args, **kwargs)	Localize tz-naive TimeSeries to target time zone.
unique()	Return unique values in the object.
unstack([level, fill_value])	Unstack, a.k.a.

Continued on next page

Table 5.47 – continued from previous page

update(other)	Modify Series in place using non-NA values from passed Series.
valid([inplace])	
value_counts([normalize, sort, ascending, ...])	Returns object containing counts of unique values.
var([axis, skipna, level, ddof, numeric_only])	Return unbiased variance over requested axis.
view([dtype])	
where(cond[, other, inplace, axis, level, ...])	Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.
xs(key[, axis, level, drop_level])	Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

T
return the transpose, which is by definition self

abs()
Return an object with absolute value taken–only applicable to objects that are all numeric.

Returns abs: type of caller

add(*other*, *level=None*, *fill_value=None*, *axis=0*)
Addition of series and other, element-wise (binary operator *add*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.radd`

add_prefix(*prefix*)
Concatenate prefix string with panel items names.

Parameters **prefix** : string

Returns **with_prefix** : type of caller

add_suffix(*suffix*)
Concatenate suffix string with panel items names.

Parameters **suffix** : string

Returns **with_suffix** : type of caller

agg(*func*, *axis=0*, **args*, ***kwargs*)
Aggregate using callable, string, dict, or list of string/callables

New in version 0.20.0.

Parameters **func** : callable, string, dictionary, or list of string/callables

Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.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)

Returns **aggregated** : Series

See also:

`pandas.Series.apply`, `pandas.Series.transform`

Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., np.mean(arr_2d, axis=0)) as opposed to mimicking the default Numpy behavior (e.g., np.mean(arr_2d)).

agg is an alias for aggregate. Use it.

Examples

```
>>> s = Series(np.random.randn(10))
```

```
>>> s.agg('min')
-1.3018049988556679
```

```
>>> s.agg(['min', 'max'])
min   -1.301805
max    1.127688
dtype: float64
```

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

Aggregate using callable, string, dict, or list of string/callables

New in version 0.20.0.

Parameters **func** : callable, string, dictionary, or list of string/callables

Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.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)

Returns **aggregated** : Series

See also:

`pandas.Series.apply`, `pandas.Series.transform`

Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0 (e.g., np.mean(arr_2d, axis=0)) as opposed to mimicking the default Numpy behavior (e.g., np.mean(arr_2d)).

agg is an alias for aggregate. Use it.

Examples

```
>>> s = Series(np.random.randn(10))
```

```
>>> s.agg('min')
-1.3018049988556679
```

```
>>> s.agg(['min', 'max'])
min   -1.301805
max    1.127688
dtype: float64
```

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 object on their axes with the specified join method for each axis Index

Parameters **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, 'index'}, default 0

Filling axis, method and limit

broadcast_axis : {0, 'index'}, default None

Broadcast values along this axis, if aligning two objects of different dimensions

New in version 0.17.0.

Returns (left, right) : (Series, type of other)

Aligned objects

all(*axis=None*, *bool_only=None*, *skipna=None*, *level=None*, ***kwargs*)
Return whether all elements are True over requested axis

Parameters axis : {index (0)}

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 scalar

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.

Returns all : scalar or Series (if level specified)

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

Parameters axis : {index (0)}

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 scalar

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.

Returns any : scalar or Series (if level specified)

append(*to_append*, *ignore_index=False*, *verify_integrity=False*)
Concatenate two or more Series.

Parameters to_append : Series or list/tuple of Series

ignore_index : boolean, default False

If True, do not use the index labels.

verify_integrity : boolean, default False

If True, raise Exception on creating index with duplicates

Returns appended : Series

Examples

```
>>> s1 = pd.Series([1, 2, 3])
>>> s2 = pd.Series([4, 5, 6])
>>> s3 = pd.Series([4, 5, 6], index=[3,4,5])
>>> s1.append(s2)
0    1
1    2
2    3
0    4
1    5
2    6
dtype: int64
```

```
>>> s1.append(s3)
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

With *ignore_index* set to True:

```
>>> s1.append(s2, ignore_index=True)
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

With *verify_integrity* set to True:

```
>>> s1.append(s2, verify_integrity=True)
Traceback (most recent call last):
...
ValueError: Indexes have overlapping values: [0, 1, 2]
```

apply (*func*, *convert_dtype=True*, *args=()*, ***kwds*)

Invoke function on values of Series. Can be ufunc (a NumPy function that applies to the entire Series) or a Python function that only works on single values

Parameters **func** : function

convert_dtype : boolean, default True

Try to find better dtype for elementwise function results. If False, leave as dtype=object

args : tuple

Positional arguments to pass to function in addition to the value

Additional keyword arguments will be passed as keywords to the function

Returns **y** : Series or DataFrame if func returns a Series

See also:

Series.map For element-wise operations

Series.agg only perform aggregating type operations

Series.transform only perform transformating type operations

Examples

Create a series with typical summer temperatures for each city.

```
>>> import pandas as pd
>>> import numpy as np
>>> series = pd.Series([20, 21, 12], index=['London',
... 'New York','Helsinki'])
>>> series
London      20
New York    21
Helsinki    12
dtype: int64
```

Square the values by defining a function and passing it as an argument to `apply()`.

```
>>> def square(x):
...     return x**2
>>> series.apply(square)
London      400
New York    441
Helsinki    144
dtype: int64
```

Square the values by passing an anonymous function as an argument to `apply()`.

```
>>> series.apply(lambda x: x**2)
London      400
New York    441
Helsinki    144
dtype: int64
```

Define a custom function that needs additional positional arguments and pass these additional arguments using the `args` keyword.

```
>>> def subtract_custom_value(x, custom_value):
...     return x-custom_value
```

```
>>> series.apply(subtract_custom_value, args=(5,))
London      15
New York    16
Helsinki     7
dtype: int64
```

Define a custom function that takes keyword arguments and pass these arguments to `apply`.

```
>>> def add_custom_values(x, **kwargs):
...     for month in kwargs:
```

```
...         x+=kwargs[month]
...         return x
```

```
>>> series.apply(add_custom_values, june=30, july=20, august=25)
London      95
New York    96
Helsinki    87
dtype: int64
```

Use a function from the Numpy library.

```
>>> series.apply(np.log)
London      2.995732
New York    3.044522
Helsinki    2.484907
dtype: float64
```

argmax (*axis=None*, *skipna=True*, **args*, ***kwargs*)
Index of first occurrence of maximum of values.

Parameters **skipna** : boolean, default True

Exclude NA/null values

Returns **idxmax** : Index of maximum of values

See also:

`DataFrame.idxmax`, `numpy.ndarray.argmax`

Notes

This method is the Series version of `ndarray.argmax`.

argmin (*axis=None*, *skipna=True*, **args*, ***kwargs*)
Index of first occurrence of minimum of values.

Parameters **skipna** : boolean, default True

Exclude NA/null values

Returns **idxmin** : Index of minimum of values

See also:

`DataFrame.idxmin`, `numpy.ndarray.argmin`

Notes

This method is the Series version of `ndarray.argmin`.

argsort (*axis=0*, *kind='quicksort'*, *order=None*)
Overrides ndarray.argsort. Argsorts the value, omitting NA/null values, and places the result in the same locations as the non-NA values

Parameters **axis** : int (can only be zero)

kind : {'mergesort', 'quicksort', 'heapsort'}, default 'quicksort'

Choice of sorting algorithm. See np.sort for more information. 'mergesort' is the only stable algorithm

order : ignored

Returns **argsorted** : Series, with -1 indicated where nan values are present

See also:

`numpy.ndarray.argsort`

as_blocks(*copy=True*)

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

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

Parameters **copy** : boolean, default True

Returns **values** : a dict of dtype -> Constructor Types

as_matrix(*columns=None*)

Convert the frame to its Numpy-array representation.

Parameters **columns: list, optional, default:None**

If None, return all columns, otherwise, returns specified columns.

Returns **values** : ndarray

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

See also:

`pandas.DataFrame.values`

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'.

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.

Parameters **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.

Returns converted : type of caller

See also:

reindex

Notes

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

Examples

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
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
```

asobject
return object Series which contains boxed values

this is an internal non-public method

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

Parameters **where** : date or array of dates

subset : string or list of strings, default None

if not None use these columns for NaN propagation

Returns where is scalar

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

where is Index: same shape object as input

See also:

`merge_asof`

Notes

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

astype(**args*, ***kwargs*)
Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

Parameters **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.

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 : DEPRECATED use `errors` instead

kwargs : keyword arguments to pass on to the constructor

Returns **casted** : type of caller

at

Fast label-based scalar accessor

Similarly to `loc`, `at` provides **label** based scalar lookups. You can also set using these indexers.

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

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

Parameters **time** : datetime.time or string

Returns **values_at_time** : type of caller

autocorr(*lag=1*)

Lag-N autocorrelation

Parameters **lag** : int, default 1

Number of lags to apply before performing autocorrelation.

Returns **autocorr** : float

axes

Return a list of the row axis labels

base

return the base object if the memory of the underlying data is shared

between(*left*, *right*, *inclusive=True*)

Return boolean Series equivalent to left <= series <= right. NA values will be treated as False

Parameters **left** : scalar

Left boundary

right : scalar

Right boundary

Returns **is_between** : Series

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).

Parameters **start_time** : datetime.time or string

end_time : datetime.time or string

include_start : boolean, default True

include_end : boolean, default True

Returns **values_between_time** : type of caller

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

blocks
Internal property, property synonym for as_blocks()

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

calcfp(**args*, ***kwargs*)
Helper function to map FP calculation throuugh the series

cat
alias of `CategoricalAccessor`

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

Parameters **lower** : float or array_like, default None

upper : float or array_like, default None

axis : int or string axis name, optional

Align object with lower and upper along the given axis.

Returns **clipped** : Series

Examples

```
>>> df
  0         1
0  0.335232 -1.256177
1 -1.367855  0.746646
2  0.027753 -1.176076
3  0.230930 -0.679613
4  1.261967  0.570967
>>> df.clip(-1.0, 0.5)
          0         1
0  0.335232 -1.000000
1 -1.000000  0.500000
2  0.027753 -1.000000
3  0.230930 -0.679613
4  0.500000  0.500000
>>> t
0   -0.3
1   -0.2
2   -0.1
3    0.0
4    0.1
dtype: float64
>>> df.clip(t, t + 1, axis=0)
```

```
          0         1
0  0.335232 -0.300000
1 -0.200000  0.746646
2  0.027753 -0.100000
3  0.230930  0.000000
4  1.100000  0.570967
```

clip_lower(*threshold*, *axis=None*)
Return copy of the input with values below given value(s) truncated.

Parameters **threshold** : float or array_like

axis : int or string axis name, optional

Align object with threshold along the given axis.

Returns **clipped** : same type as input

See also:

clip

clip_upper(*threshold*, *axis=None*)
Return copy of input with values above given value(s) truncated.

Parameters **threshold** : float or array_like

axis : int or string axis name, optional

Align object with threshold along the given axis.

Returns **clipped** : same type as input

See also:

clip

combine(*other*, *func*, *fill_value=nan*)
Perform elementwise binary operation on two Series using given function with optional fill value when an index is missing from one Series or the other

Parameters **other** : Series or scalar value

func : function

fill_value : scalar value

Returns **result** : Series

combine_first(*other*)
Combine Series values, choosing the calling Series's values first. Result index will be the union of the two indexes

Parameters **other** : Series

Returns **y** : Series

compound(*axis=None*, *skipna=None*, *level=None*)
Return the compound percentage of the values for the requested axis

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **compounded** : scalar or Series (if level specified)

compress(*condition*, **args*, ***kwargs*)

Return selected slices of an array along given axis as a Series

See also:

`numpy.ndarray.compress`

consolidate(*inplace=False*)

DEPRECATED: consolidate will be an internal implementation only.

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

Deprecated.

Attempt to infer better dtype for object columns

Parameters **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.

Returns **converted** : same as input object

See also:

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.

copy(*deep=True*)

Make a copy of this objects data.

Parameters **deep** : boolean or string, default True

Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices or the data are copied.

Note that when `deep=True` data is copied, 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.

Returns **copy** : type of caller

corr(*other*, *method='pearson'*, *min_periods=None*)

Compute correlation with *other* Series, excluding missing values

Parameters **other** : Series

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 needed to have a valid result

Returns **correlation** : float

count(*level=None*)

Return number of non-NA/null observations in the Series

Parameters **level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

Returns **nobs** : int or Series (if level specified)

cov(*other*, *min_periods=None*)

Compute covariance with Series, excluding missing values

Parameters **other** : Series

min_periods : int, optional

Minimum number of observations needed to have a valid result

Returns **covariance** : float

Normalized by N-1 (unbiased estimator).

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

Return cumulative max over requested axis.

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cummax** : scalar

See also:

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

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

Return cumulative minimum over requested axis.

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cummin** : scalar

See also:

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

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

Return cumulative product over requested axis.

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cumprod** : scalar

See also:

pandas.core.window.Expanding.prod Similar functionality but ignores `NaN` values.

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

Return cumulative sum over requested axis.

Parameters **axis** : {index (0)}

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns **cumsum** : scalar

See also:

pandas.core.window.Expanding.sum Similar functionality but ignores `NaN` values.

data

return the data pointer of the underlying data

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.

Parameters **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 categorical objects submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`)
- 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 select numeric types submit `numpy.number`. To select categorical objects submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`)
- None (default) : The result will exclude nothing.

Returns summary: Series/DataFrame of summary statistics

See also:

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

Notes

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 `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`.

Examples

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([[1, 'a'], [2, 'b'], [3, 'c']],
...                   columns=['numeric', 'object'])
>>> 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')
        numeric object
count       3.0      3
unique      NaN      3
top         NaN      b
freq        NaN      1
mean        2.0    NaN
std         1.0    NaN
min         1.0    NaN
25%         1.5    NaN
50%         2.0    NaN
```

```
75%          2.5     NaN
max          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         b
freq        1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       object
count       3
unique      3
top         b
freq        1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.object])
       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
```

diff (*periods=1*)
1st discrete difference of object

Parameters **periods** : int, default 1

Periods to shift for forming difference

Returns **diffed** : Series

div (*other*, *level=None*, *fill_value=None*, *axis=0*)
Floating division of series and other, element-wise (binary operator *truediv*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.rtruediv`

divide (*other*, *level=None*, *fill_value=None*, *axis=0*)
Floating division of series and other, element-wise (binary operator *truediv*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.rtruediv`

dot (*other*)
Matrix multiplication with DataFrame or inner-product with Series objects

Parameters **other** : Series or DataFrame

Returns **dot_product** : scalar or Series

drop (*labels*, *axis=0*, *level=None*, *inplace=False*, *errors='raise'*)
Return new object with labels in requested axis removed.

Parameters **labels** : single label or list-like

axis : int or axis name

level : int or level name, default None

For MultiIndex

inplace : bool, default False

If True, do operation inplace and return None.

errors : {'ignore', 'raise'}, default 'raise'

If 'ignore', suppress error and existing labels are dropped.

New in version 0.16.1.

Returns **dropped** : type of caller

drop_duplicates(*keep='first'*, *inplace=False*)

Return Series with duplicate values removed

Parameters **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

If True, performs operation inplace and returns None.

Returns **deduplicated** : Series

dropna(*axis=0*, *inplace=False*, ***kwargs*)

Return Series without null values

Returns **valid** : Series

inplace : boolean, default False

Do operation in place.

dt

alias of `CombinedDatetimelikeProperties`

dtype

return the dtype object of the underlying data

dtypes

return the dtype object of the underlying data

duplicated(*keep='first'*)

Return boolean Series denoting duplicate values

Parameters **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`.

Returns **duplicated** : Series

empty

eq(*other*, *level=None*, *fill_value=None*, *axis=0*)
Equal to of series and other, element-wise (binary operator *eq*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.None`

equals(*other*)
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

ewm(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *freq=None*, *adjust=True*, *ignore_na=False*, *axis=0*)
Provides exponential weighted functions

New in version 0.18.0.

Parameters **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).

freq : None or string alias / date offset object, default=None (DEPRECATED)

Frequency to conform to before computing statistic

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

Returns a Window sub-classed for the particular operation

Notes

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.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

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

Examples

```
>>> 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
```

expanding (*min_periods=1*, *freq=None*, *center=False*, *axis=0*)

Provides expanding transformations.

New in version 0.18.0.

Parameters **min_periods** : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA).

freq : string or DateOffset object, optional (default None) (DEPRECATED)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

axis : int or string, default 0

Returns a Window sub-classed for the particular operation

Notes

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`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

Examples

```
>>> 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
```

factorize(*sort=False*, *na_sentinel=-1*)

Encode the object as an enumerated type or categorical variable

Parameters **sort** : boolean, default False

Sort by values

na_sentinel: int, default -1

Value to mark "not found"

Returns **labels** : the indexer to the original array

uniques : the unique Index

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

Parameters **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, 'index'}

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)

Returns **filled** : Series

See also:

reindex, *asfreq*

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.

Parameters **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

Returns same type as input object

See also:

`pandas.DataFrame.select`

Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

Examples

```
>>> 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
```

first (*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.first('10D') -> First 10 days

first_valid_index ()

Return label for first non-NA/null value

flags

return the ndarray.flags for the underlying data

floordiv (*other*, *level=None*, *fill_value=None*, *axis=0*)

Integer division of series and other, element-wise (binary operator *floordiv*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.rfloordiv`

from_array(*arr*, *index=None*, *name=None*, *dtype=None*, *copy=False*, *fastpath=False*)

from_csv(*path*, *sep=','*, *','*, *parse_dates=True*, *header=None*, *index_col=0*, *encoding=None*, *infer_datetime_format=False*)

Read CSV file (DISCOURAGED, please 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 time Series.

This method only differs from `pandas.read_csv()` in some defaults:

- *index_col* is `0` instead of `None` (take first column as index by default)
- *header* is `None` instead of `0` (the first row is not used as the column names)
- *parse_dates* is `True` instead of `False` (try parsing the index as datetime by default)

With `pandas.read_csv()`, the option `squeeze=True` can be used to return a Series like `from_csv`.

Parameters **path** : string file path or file handle / StringIO

sep : string, default ','

Field delimiter

parse_dates : boolean, default True

Parse dates. Different default from read_table

header : int, default None

Row to use as header (skip prior rows)

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

encoding : string, optional

a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3

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.

Returns **y** : Series

See also:

`pandas.read_csv`

ftype
return if the data is sparseldense

ftypes
return if the data is sparseldense

ge(*other*, *level=None*, *fill_value=None*, *axis=0*)
Greater than or equal to of series and other, element-wise (binary operator *ge*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.None`

get(*key*, *default=None*)
Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

Parameters **key** : object

Returns **value** : type of items contained in object

get_dtype_counts()
Return the counts of dtypes in this object.

get_ftype_counts()
Return the counts of ftypes in this object.

get_value(*label*, *takeable=False*)
Quickly retrieve single value at passed index label

Parameters **index** : label

takeable : interpret the index as indexers, default False

Returns **value** : scalar value

get_values()
same as values (but handles sparseness conversions); is a view

groupby(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=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.

Parameters **by** : mapping, function, str, or iterable

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 str or list of strs may be passed to group by the columns in `self`

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

Returns GroupBy object

Examples

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

gt(*other*, *level=None*, *fill_value=None*, *axis=0*)

Greater than of series and other, element-wise (binary operator *gt*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.None`

hasnans = None

head(*n=5*)
Returns first n rows

hist(*by=None*, *ax=None*, *grid=True*, *xlabelsize=None*, *xrot=None*, *ylabelsize=None*, *yrot=None*, *figsize=None*, *bins=10*, ***kwds*)
Draw histogram of the input series using matplotlib

Parameters **by** : object, optional

If passed, then used to form histograms for separate groups

ax : matplotlib axis object

If not passed, uses gca()

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

ylabelsize : int, default None

If specified changes the y-axis label size

yrot : float, default None

rotation of y axis labels

figsize : tuple, default None

figure size in inches by default

bins: integer, default 10

Number of histogram bins to be used

kwds : keywords

To be passed to the actual plotting function

Notes

See matplotlib documentation online for more on this

iat
Fast integer location scalar accessor.

Similarly to `iloc`, `iat` provides **integer** based lookups. You can also set using these indexers.

idxmax(*axis=None*, *skipna=True*, **args*, ***kwargs*)
Index of first occurrence of maximum of values.

Parameters **skipna** : boolean, default True

Exclude NA/null values

Returns **idxmax** : Index of maximum of values

See also:

`DataFrame.idxmax`, `numpy.ndarray.argmax`

Notes

This method is the Series version of `ndarray.argmax`.

idxmin(*axis=None*, *skipna=True*, **args*, ***kwargs*)
Index of first occurrence of minimum of values.

Parameters **skipna** : boolean, default True

Exclude NA/null values

Returns **idxmin** : Index of minimum of values

See also:

`DataFrame.idxmin`, `numpy.ndarray.argmin`

Notes

This method is the Series version of `ndarray.argmin`.

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

imag

interpolate(*method='linear'*, *axis=0*, *limit=None*, *inplace=False*, *limit_direction='forward'*, *downcast=None*, ***kwargs*)
Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

Parameters **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'

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.17.0.

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.

Returns Series or DataFrame of same shape interpolated at the NaNs

See also:

reindex, *replace*, *fillna*

Examples

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 = None

is_monotonic

Return boolean if values in the object are monotonic_increasing

New in version 0.19.0.

Returns **is_monotonic** : boolean

is_monotonic_decreasing

Return boolean if values in the object are monotonic_decreasing

New in version 0.19.0.

Returns **is_monotonic_decreasing** : boolean

is_monotonic_increasing

Return boolean if values in the object are monotonic_increasing

New in version 0.19.0.

Returns **is_monotonic** : boolean

is_unique

Return boolean if values in the object are unique

Returns **is_unique** : boolean

isin(*values*)

Return a boolean `Series` showing whether each element in the `Series` is exactly contained in the passed sequence of `values`.

Parameters **values** : set or list-like

The sequence of values to test. Passing in a single string will raise a `TypeError`. Instead, turn a single string into a `list` of one element.

New in version 0.18.1.

Support for values as a set

Returns **isin** : Series (bool dtype)

Raises **TypeError**

- If `values` is a string

See also:

`pandas.DataFrame.isin`

Examples

```
>>> s = pd.Series(list('abc'))
>>> s.isin(['a', 'c', 'e'])
0     True
1    False
2     True
dtype: bool
```

Passing a single string as `s.isin('a')` will raise an error. Use a list of one element instead:

```
>>> s.isin(['a'])
0     True
1    False
2    False
dtype: bool
```

isnull()
Return a boolean same-sized object indicating if the values are null.

See also:

notnull boolean inverse of isnull

item()
return the first element of the underlying data as a python scalar

itemsize
return the size of the dtype of the item of the underlying data

iteritems()
Lazily iterate over (index, value) tuples

ix
A primarily label-location based indexer, with integer position fallback.

`.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 hierachical 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.

keys()
Alias for index

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

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **kurt** : scalar or Series (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

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **kurt** : scalar or Series (if level specified)

last(*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

Parameters **offset** : string, DateOffset, dateutil.relativedelta

Returns **subset** : type of caller

Examples

ts.last('5M') -> Last 5 months

last_valid_index()

Return label for last non-NA/null value

le(*other*, *level=None*, *fill_value=None*, *axis=0*)

Less than or equal to of series and other, element-wise (binary operator *le*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.None`

loc

Purely label-location based indexer for selection by label.

`.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'` (note that contrary to usual python slices, **both** the start and the stop are included!).
- 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)

`.loc` will raise a `KeyError` when the items are not found.

See more at Selection by Label

lt(*other*, *level=None*, *fill_value=None*, *axis=0*)

Less than of series and other, element-wise (binary operator *lt*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.None`

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

Return the mean absolute deviation of the values for the requested axis

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **mad** : scalar or Series (if level specified)

map(*arg*, *na_action=None*)

Map values of Series using input correspondence (which can be a dict, Series, or function)

Parameters **arg** : function, dict, or Series

na_action : {None, 'ignore'}

If 'ignore', propagate NA values, without passing them to the mapping function

Returns **y** : Series

same index as caller

See also:

Series.apply For applying more complex functions on a Series

DataFrame.apply Apply a function row-/column-wise

DataFrame.applymap Apply a function elementwise on a whole DataFrame

Notes

When *arg* is a dictionary, values in Series that are not in the dictionary (as keys) are converted to `NaN`. However, if the dictionary is a `dict` subclass that defines `__missing__` (i.e. provides a method for default values), then this default is used rather than `NaN`:

```
>>> from collections import Counter
>>> counter = Counter()
>>> counter['bar'] += 1
>>> y.map(counter)
1    0
2    1
3    0
dtype: int64
```

Examples

Map inputs to outputs (both of type *Series*)

```
>>> x = pd.Series([1,2,3], index=['one', 'two', 'three'])
>>> x
one      1
two      2
three    3
dtype: int64
```

```
>>> y = pd.Series(['foo', 'bar', 'baz'], index=[1,2,3])
>>> y
1    foo
2    bar
3    baz
```

```
>>> x.map(y)
one   foo
two   bar
three baz
```

If *arg* is a dictionary, return a new Series with values converted according to the dictionary's mapping:

```
>>> z = {1: 'A', 2: 'B', 3: 'C'}
```

```
>>> x.map(z)
one   A
two   B
three C
```

Use na_action to control whether NA values are affected by the mapping function.

```
>>> s = pd.Series([1, 2, 3, np.nan])
```

```
>>> s2 = s.map('this is a string {}'.format, na_action=None)
0    this is a string 1.0
1    this is a string 2.0
2    this is a string 3.0
3    this is a string nan
dtype: object
```

```
>>> s3 = s.map('this is a string {}'.format, na_action='ignore')
0    this is a string 1.0
1    this is a string 2.0
2    this is a string 3.0
3                    NaN
dtype: object
```

mask(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *try_cast=False*, *raise_on_error=True*)

Return an object of same shape as self and whose corresponding entries are from self where cond is False and otherwise are from other.

Parameters **cond** : boolean NDFrame, array-like, or callable

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

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

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)

Returns **wh** : same type as caller

See also:

`DataFrame.where()`

Notes

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
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
```

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`.

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **max** : scalar or Series (if level specified)

mean(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the mean of the values for the requested axis

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **mean** : scalar or Series (if level specified)

median(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the median of the values for the requested axis

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **median** : scalar or Series (if level specified)

memory_usage(*index=True*, *deep=False*)
Memory usage of the Series

Parameters **index** : bool

Specifies whether to include memory usage of Series index

deep : bool

Introspect the data deeply, interrogate *object* dtypes for system-level memory consumption

Returns scalar bytes of memory consumed

See also:

`numpy.ndarray.nbytes`

Notes

Memory usage does not include memory consumed by elements that are not components of the array if deep=False

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`.

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **min** : scalar or Series (if level specified)

mod(*other*, *level=None*, *fill_value=None*, *axis=0*)
Modulo of series and other, element-wise (binary operator *mod*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.rmod`

mode()
Return the mode(s) of the dataset.

Always returns Series even if only one value is returned.

Returns **modes** : Series (sorted)

mul(*other*, *level=None*, *fill_value=None*, *axis=0*)
Multiplication of series and other, element-wise (binary operator *mul*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.rmul`

multiply(*other*, *level=None*, *fill_value=None*, *axis=0*)
Multiplication of series and other, element-wise (binary operator *mul*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.rmul`

name

nbytes
return the number of bytes in the underlying data

ndim
return the number of dimensions of the underlying data, by definition 1

ne(*other*, *level=None*, *fill_value=None*, *axis=0*)
Not equal to of series and other, element-wise (binary operator *ne*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.None`

nlargest (*n=5*, *keep='first'*)

Return the largest *n* elements.

Parameters **n** : int

Return this many descending sorted values

keep : {'first', 'last', False}, default 'first'

Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

Returns **top_n** : Series

The n largest values in the Series, in sorted order

See also:

`Series.nsmallest`

Notes

Faster than `.sort_values(ascending=False).head(n)` for small *n* relative to the size of the `Series` object.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(10**6))
>>> s.nlargest(10)  # only sorts up to the N requested
219921    4.644710
82124     4.608745
421689    4.564644
425277    4.447014
718691    4.414137
43154     4.403520
283187    4.313922
595519    4.273635
503969    4.250236
```

```
121637    4.240952
dtype: float64
```

nonzero()
Return the indices of the elements that are non-zero

This method is equivalent to calling *numpy.nonzero* on the series data. For compatability with NumPy, the return value is the same (a tuple with an array of indices for each dimension), but it will always be a one-item tuple because series only have one dimension.

See also:

`numpy.nonzero`

Examples

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

notnull()
Return a boolean same-sized object indicating if the values are not null.

See also:

isnull boolean inverse of notnull

nsmallest(*n=5*, *keep='first'*)
Return the smallest *n* elements.

Parameters **n** : int

Return this many ascending sorted values

keep : {'first', 'last', False}, default 'first'

Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

Returns **bottom_n** : Series

The n smallest values in the Series, in sorted order

See also:

`Series.nlargest`

Notes

Faster than `.sort_values().head(n)` for small *n* relative to the size of the `Series` object.

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(10**6))
>>> s.nsmallest(10)  # only sorts up to the N requested
288532   -4.954580
732345   -4.835960
64803    -4.812550
446457   -4.609998
501225   -4.483945
669476   -4.472935
973615   -4.401699
621279   -4.355126
773916   -4.347355
359919   -4.331927
dtype: float64
```

nunique (*dropna=True*)
Return number of unique elements in the object.

Excludes NA values by default.

Parameters **dropna** : boolean, default True

Don't include NaN in the count.

Returns **nunique** : int

pct_change (*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)
Percent change over given number of periods.

Parameters **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())

Returns **chg** : NDFrame

Notes

By default, the percentage change is calculated along the stat axis: 0, or `Index`, for `DataFrame` and 1, or `minor` for `Panel`. You can change this with the `axis` keyword argument.

pipe (*func*, **args*, ***kwargs*)
Apply func(self, *args, **kwargs)

New in version 0.16.2.

Parameters **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 : positional arguments passed into `func`.

kwargs : a dictionary of keyword arguments passed into `func`.

Returns **object** : the return type of `func`.

See also:

`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`

Notes

Use `.pipe` when chaining together functions that expect on Series or DataFrames. 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)
...  )
```

plot
alias of `SeriesPlotMethods`

pop(*item*)
Return item and drop from frame. Raise KeyError if not found.

pow(*other*, *level=None*, *fill_value=None*, *axis=0*)
Exponential power of series and other, element-wise (binary operator *pow*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.rpow`

prod(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the product of the values for the requested axis

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **prod** : scalar or Series (if level specified)

product(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the product of the values for the requested axis

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **prod** : scalar or Series (if level specified)

ptp(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Returns the difference between the maximum value and the minimum value in the object. This is the equivalent of the `numpy.ndarray` method `ptp`.

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **ptp** : scalar or Series (if level specified)

put (**args*, ***kwargs*)
Applies the *put* method to its *values* attribute if it has one.

See also:

`numpy.ndarray.put`

quantile (*q=0.5*, *interpolation='linear'*)
Return value at the given quantile, a la numpy.percentile.

Parameters **q** : float or array-like, default 0.5 (50% quantile)

0 <= q <= 1, the quantile(s) to compute

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.

Returns **quantile** : float or Series

if `q` is an array, a Series will be returned where the index is `q` and the values are the quantiles.

Examples

```
>>> s = Series([1, 2, 3, 4])
>>> s.quantile(.5)
2.5
>>> s.quantile([.25, .5, .75])
0.25    1.75
0.50    2.50
0.75    3.25
dtype: float64
```

radd (*other*, *level=None*, *fill_value=None*, *axis=0*)
Addition of series and other, element-wise (binary operator *radd*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.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

Parameters **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

Returns **ranks** : same type as caller

ravel(*order='C'*)

Return the flattened underlying data as an ndarray

See also:

`numpy.ndarray.ravel`

rdiv(*other*, *level=None*, *fill_value=None*, *axis=0*)

Floating division of series and other, element-wise (binary operator *rtruediv*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.truediv`

real

reindex(*index=None*, ***kwargs*)

Conform Series 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

Parameters **index** : array-like, optional (can be specified in order, or as

keywords) New labels / index to conform to. Preferably an Index object to avoid duplicating data

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`.

New in version 0.17.0.

Returns **reindexed** : Series

Examples

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
IE10                  404          0.08
Chrome                200          0.02
```

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.

reindex_axis (*labels*, *axis=0*, ***kwargs*)
for compatibility with higher dims

reindex_like (*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)
Return an object with matching indices to myself.

Parameters **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.

New in version 0.17.0.

Returns **reindexed** : same as input

Notes

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

rename(*index=None*, ***kwargs*)

Alter axes input function or functions. 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. Alternatively, change `Series.name` with a scalar value (Series only).

Parameters **index** : scalar, list-like, dict-like or function, optional

Scalar or list-like will alter the `Series.name` attribute, and raise on DataFrame or Panel. dict-like or functions are transformations to apply to that axis' values

copy : boolean, default True

Also copy underlying data

inplace : boolean, default False

Whether to return a new Series. 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.

Returns **renamed** : Series (new object)

See also:

`pandas.NDFrame.rename_axis`

Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
>>> s.rename("my_name") # scalar, changes Series.name
0    1
1    2
2    3
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2)  # function, changes labels
0    1
1    2
4    3
dtype: int64
>>> s.rename({1: 3, 2: 5})  # mapping, changes labels
0    1
```

```
3    2
5    3
dtype: int64
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(2)
Traceback (most recent call last):
...
TypeError: 'int' object is not callable
>>> 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
1  2  5
2  3  6
```

rename_axis(*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter index and / or columns using input function or functions. A scalar or list-like for `mapper` will alter the `Index.name` or `MultiIndex.names` attribute. A function or dict for `mapper` will alter the labels. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is.

Parameters **mapper** : scalar, list-like, dict-like or function, optional

axis : int or string, default 0

copy : boolean, default True

Also copy underlying data

inplace : boolean, default False

Returns **renamed** : type of caller

See also:

`pandas.NDFrame.rename`, `pandas.Index.rename`

Examples

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")  # scalar, alters df.index.name
     A  B
foo
0    1  4
1    2  5
2    3  6
>>> df.rename_axis(lambda x: 2 * x)  # function: alters labels
   A  B
0  1  4
2  2  5
4  3  6
>>> df.rename_axis({"A": "ehh", "C": "see"}, axis="columns")  # mapping
   ehh  B
0    1  4
```

```
1    2  5
2    3  6
```

reorder_levels(*order*)

Rearrange index levels using input order. May not drop or duplicate levels

Parameters **order** : list of int representing new level order.

(reference level by number or key)

axis : where to reorder levels

Returns type of caller (new object)

repeat(**args*, ***kwargs*)

Repeat elements of an Series. Refer to *numpy.ndarray.repeat* for more information about the *repeats* argument.

See also:

`numpy.ndarray.repeat`

replace(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*, *axis=None*)

Replace values given in 'to_replace' with 'value'.

Parameters **to_replace** : str, regex, list, dict, Series, numeric, or None

- str or regex:
  - 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 and regex rules apply as above.
- dict:
  - Nested dictionaries, e.g., {'a': {'b': nan}}, are read as follows: look in column 'a' for the value 'b' and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
  - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
- 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 use to fill holes (e.g. 0), alternately a dict of values specifying 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 form 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. Otherwise, *to_replace* must be `None` because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.

method : string, optional, {'pad', 'ffill', 'bfill'}

The method to use when for replacement, when `to_replace` is a `list`.

Returns filled : NDFrame

Raises 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.

ValueError

- If *to_replace* and *value* are `list` s or `ndarray` s, but they are not the same length.

See also:

`NDFrame.reindex`, `NDFrame.asfreq`, `NDFrame.fillna`

Notes

- 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.

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.

Parameters **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

label : {'right', 'left'}

Which bin edge label to label bucket with

convention : {'start', 'end', 's', 'e'}

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.

Notes

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

Examples

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 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
```

reset_index(*level=None*, *drop=False*, *name=None*, *inplace=False*)

Analogous to the `pandas.DataFrame.reset_index()` function, see docstring there.

Parameters **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

name : object, default None

The name of the column corresponding to the Series values

inplace : boolean, default False

Modify the Series in place (do not create a new object)

Returns **resetted** : DataFrame, or Series if drop == True

reshape(**args*, ***kwargs*)
DEPRECATED: calling this method will raise an error in a future release. Please call `.values.reshape(...)` instead.

return an ndarray with the values shape if the specified shape matches exactly the current shape, then return self (for compat)

See also:

`numpy.ndarray.reshape`

rfloordiv(*other*, *level=None*, *fill_value=None*, *axis=0*)
Integer division of series and other, element-wise (binary operator *rfloordiv*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.floordiv`

rmod(*other*, *level=None*, *fill_value=None*, *axis=0*)
Modulo of series and other, element-wise (binary operator *rmod*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.mod`

rmul(*other*, *level=None*, *fill_value=None*, *axis=0*)
Multiplication of series and other, element-wise (binary operator *rmul*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.mul`

rolling(*window*, *min_periods=None*, *freq=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculcations.

New in version 0.18.0.

Parameters **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.

freq : string or DateOffset object, optional (default None) (DEPRECATED)

Frequency to conform the data to before computing the statistic. Specified as a frequency string or DateOffset object.

center : boolean, default False

Set the labels at the center of the window.

win_type : string, default None

Provide a window type. See the notes below.

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

Returns a Window or Rolling sub-classed for the particular operation

Notes

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`.

The *freq* keyword is used to conform time series data to a specified frequency by resampling the data. This is done with the default parameters of `resample()` (i.e. using the *mean*).

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).

Examples

```
>>> 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
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 explicity 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
```

round(*decimals=0*, **args*, ***kwargs*)

Round each value in a Series to the given number of decimals.

Parameters **decimals** : int

Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

Returns Series object

See also:

`numpy.around`, `DataFrame.round`

rpow(*other*, *level=None*, *fill_value=None*, *axis=0*)
Exponential power of series and other, element-wise (binary operator *rpow*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.pow`

rsub(*other*, *level=None*, *fill_value=None*, *axis=0*)
Subtraction of series and other, element-wise (binary operator *rsub*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.sub`

rtruediv(*other*, *level=None*, *fill_value=None*, *axis=0*)
Floating division of series and other, element-wise (binary operator *rtruediv*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.truediv`

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*)
Returns a random sample of items from an axis of object.

New in version 0.16.1.

Parameters **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).

Returns A new object of same type as caller.

Examples

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
```

searchsorted(**args*, ***kwargs*)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted Series *self* such that, if the corresponding elements in *value* were inserted before the indices, the order of *self* would be preserved.

Parameters **value** : array_like

Values to insert into *self*.

side : {'left', 'right'}, optional

If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

sorter : 1-D array_like, optional

Optional array of integer indices that sort *self* into ascending order. They are typically the result of `np.argsort`.

Returns **indices** : array of ints

Array of insertion points with the same shape as *value*.

See also:

`numpy.searchsorted`

Notes

Binary search is used to find the required insertion points.

Examples

```
>>> x = pd.Series([1, 2, 3])
>>> x
0    1
1    2
```

```
2    3
dtype: int64
```

```
>>> x.searchsorted(4)
array([3])
```

```
>>> x.searchsorted([0, 4])
array([0, 3])
```

```
>>> x.searchsorted([1, 3], side='left')
array([0, 2])
```

```
>>> x.searchsorted([1, 3], side='right')
array([1, 3])
```

```
>>> x = pd.Categorical(['apple', 'bread', 'bread', 'cheese', 'milk' ])
[apple, bread, bread, cheese, milk]
Categories (4, object): [apple < bread < cheese < milk]
```

```
>>> x.searchsorted('bread')
array([1])      # Note: an array, not a scalar
```

```
>>> x.searchsorted(['bread'])
array([1])
```

```
>>> x.searchsorted(['bread', 'eggs'])
array([1, 4])
```

```
>>> x.searchsorted(['bread', 'eggs'], side='right')
array([3, 4])     # eggs before milk
```

select (*crit*, *axis=0*)

Return data corresponding to axis labels matching criteria

Parameters **crit** : function

To be called on each index (label). Should return True or False

axis : int

Returns **selection** : type of caller

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

Parameters **axis** : {index (0)}

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 scalar

ddof : int, default 1

degrees of freedom

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.

Returns **sem** : scalar or Series (if level specified)

set_axis (*axis*, *labels*)
public verson of axis assignment

set_value (*label*, *value*, *takeable=False*)
Quickly set single value at passed label. If label is not contained, a new object is created with the label placed at the end of the result index

Parameters **label** : object

Partial indexing with MultiIndex not allowed

value : object

Scalar value

takeable : interpret the index as indexers, default False

Returns **series** : Series

If label is contained, will be reference to calling Series, otherwise a new object

shape
return a tuple of the shape of the underlying data

shift (*periods=1*, *freq=None*, *axis=0*)
Shift index by desired number of periods with an optional time freq

Parameters **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, 'index'}

Returns **shifted** : Series

Notes

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.

size
return the number of elements in the underlying data

skew (*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis Normalized by N-1

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **skew** : scalar or Series (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.

Parameters **periods** : int

Number of periods to move, can be positive or negative

Returns **shifted** : same type as caller

Notes

While the *slice_shift* is faster than *shift*, you may pay for it later during alignment.

sort_index(*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*)

Sort object by labels (along an axis)

Parameters **axis** : index 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

Returns **sorted_obj** : Series

sort_values(*axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)

Sort by the values along either axis

New in version 0.17.0.

Parameters **axis** : {0, 'index'}, default 0

Axis to direct sorting

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

Returns **sorted_obj** : Series

sortlevel(*level=0*, *ascending=True*, *sort_remaining=True*)

DEPRECATED: use `Series.sort_index()`

Sort Series with MultiIndex by chosen level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order)

Parameters **level** : int or level name, default None

ascending : bool, default True

Returns **sorted** : Series

See also:

`Series.sort_index`

squeeze(*axis=None*)

Squeeze length 1 dimensions.

Parameters **axis** : None, integer or string axis name, optional

The axis to squeeze if 1-sized.

New in version 0.20.0.

Returns scalar if 1-sized, else original object

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

Parameters **axis** : {index (0)}

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 scalar

ddof : int, default 1

degrees of freedom

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.

Returns **std** : scalar or Series (if level specified)

str
alias of `StringMethods`

strides
return the strides of the underlying data

sub(*other*, *level=None*, *fill_value=None*, *axis=0*)
Subtraction of series and other, element-wise (binary operator *sub*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.rsub`

subtract(*other*, *level=None*, *fill_value=None*, *axis=0*)
Subtraction of series and other, element-wise (binary operator *sub*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.rsub`

sum(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the sum of the values for the requested axis

Parameters **axis** : {index (0)}

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 scalar

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.

Returns **sum** : scalar or Series (if level specified)

swapaxes(*axis1*, *axis2*, *copy=True*)
Interchange axes and swap values axes appropriately

Returns **y** : same as input

swaplevel(*i=-2*, *j=-1*, *copy=True*)
Swap levels i and j in a MultiIndex

Parameters **i, j** : int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

Returns **swapped** : Series

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*)
Returns last n rows

take(*indices*, *axis=0*, *convert=True*, *is_copy=False*, ***kwargs*)
return Series corresponding to requested indices

Parameters **indices** : list / array of ints

convert : translate negative to positive indices (default)

Returns **taken** : Series

See also:

`numpy.ndarray.take`

to_clipboard(*excel=None*, *sep=None*, ***kwargs*)
Attempt to write text representation of object to the system clipboard This can be pasted into Excel, for example.

Parameters **excel** : boolean, defaults to True

if True, use the provided separator, writing in a csv format for allowing easy pasting into excel. if False, write a string representation of the object to the clipboard

sep : optional, defaults to tab

other keywords are passed to to_csv

Notes

Requirements for your platform

- Linux: xclip, or xsel (with gtk or PyQt4 modules)
- Windows: none

- OS X: none

to_csv(*path=None*, *index=True*, *sep=','*, *na_rep=''*, *float_format=None*, *header=False*, *index_label=None*, *mode='w'*, *encoding=None*, *date_format=None*, *decimal='.'*)

Write Series to a comma-separated values (csv) file

Parameters **path** : string or file handle, default None

File path or object, if None is provided the result is returned as a string.

na_rep : string, default ''

Missing data representation

float_format : string, default None

Format string for floating point numbers

header : boolean, default False

Write out series name

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.

mode : Python write mode, default 'w'

sep : character, default ","

Field delimiter for the output file.

encoding : string, optional

a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3

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()

Convert Series to {label -> value} dict

Returns **value_dict** : dict

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*)

Write Series to an excel sheet

New in version 0.20.0.

Parameters **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 column names. If a list of string 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.

Notes

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_frame(*name=None*)
Convert Series to DataFrame

Parameters **name** : object, default None

The passed name should substitute for the series name (if it has one).

Returns **data_frame** : DataFrame

to_hdf(*path_or_buf*, *key*, ***kwargs*)
Write the contained data to an HDF5 file using HDFStore.

Parameters **path_or_buf** : the path (string) or HDFStore object

key : string

identifier for the group in the store

mode : optional, {'a', 'w', 'r+'}, default 'a'

'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+' It is similar to 'a', but the file must already exist.

format : 'fixed(f)|table(t)', default is 'fixed'

fixed(f) [Fixed format] Fast writing/reading. Not-appendable, nor searchable

table(t) [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 : boolean, default False

For Table formats, append the input data to the existing

data_columns : list of columns, or True, default None

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 here.

Applicable only to format='table'.

complevel : int, 1-9, default 0

If a complib is specified compression will be applied where possible

complib : {'zlib', 'bzip2', 'lzo', 'blosc', None}, default None

If complevel is > 0 apply compression to objects written in the store wherever possible

fletcher32 : bool, default False

If applying compression use the fletcher32 checksum

dropna : boolean, default False.

If true, ALL nan rows will not be written to store.

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*)

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.

Parameters **path_or_buf** : the path or buffer to write the result string

if this is None, return a StringIO of the converted string

orient : string

- 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 : The number of decimal places to use when encoding

floating point values, default 10.

force_ascii : force encoded string to be ASCII, default True.

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.

Returns same type as input object with filtered info axis

See also:

`pd.read_json`

Examples

```
>>> 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 `'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 `'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 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_mol2 (*filepath_or_buffer=None*)

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.

Parameters **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_period (*freq=None*, *copy=True*)
Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

Parameters **freq** : string, default

Returns **ts** : Series with PeriodIndex

to_pickle (*path*, *compression='infer'*)
Pickle (serialize) object to input file path.

Parameters **path** : string

File path

compression : {'infer', 'gzip', 'bz2', 'xz', None}, default 'infer'

a string representing the compression to use in the output file

New in version 0.20.0.

to_sdf (*filepath_or_buffer=None*)

to_smiles (*filepath_or_buffer=None*)

to_sparse (*kind='block'*, *fill_value=None*)
Convert Series to SparseSeries

Parameters **kind** : {'block', 'integer'}

fill_value : float, defaults to NaN (missing)

Returns **sp** : SparseSeries

to_sql (*name*, *con*, *flavor=None*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*)
Write records stored in a DataFrame to a SQL database.

Parameters **name** : string

Name of SQL table

con : SQLAlchemy engine or DBAPI2 connection (legacy mode)

Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

flavor : 'sqlite', default None

DEPRECATED: this parameter will be removed in a future version, as 'sqlite' is the only supported option if SQLAlchemy is not installed.

schema : string, default None

Specify the schema (if database flavor supports this). If None, use default schema.

if_exists : {'fail', 'replace', 'append'}, default 'fail'

- 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.

index : boolean, default True

Write DataFrame index as a column.

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, default None

If not None, then rows will be written in batches of this size at a time. If None, all rows will be written at once.

dtype : dict of column name to SQL type, default None

Optional specifying the datatype for columns. The SQL type should be a SQLAlchemy type, or a string for sqlite3 fallback connection.

`to_string`(*buf=None*, *na_rep='NaN'*, *float_format=None*, *header=True*, *index=True*, *length=False*, *dtype=False*, *name=False*, *max_rows=None*)

Render a string representation of the Series

Parameters **buf** : StringIO-like, optional

buffer to write to

na_rep : string, optional

string representation of NAN to use, default 'NaN'

float_format : one-parameter function, optional

formatter function to apply to columns' elements if they are floats default None

header: boolean, default True

Add the Series header (index name)

index : bool, optional

Add index (row) labels, default True

length : boolean, default False

Add the Series length

dtype : boolean, default False

Add the Series dtype

name : boolean, default False

Add the Series name if not None

max_rows : int, optional

Maximum number of rows to show before truncating. If None, show all.

Returns **formatted** : string (if not buffer passed)

to_timestamp (*freq=None*, *how='start'*, *copy=True*)
Cast to datetimeindex of timestamps, at *beginning* of period

Parameters **freq** : string, default frequency of PeriodIndex

Desired frequency

how : {'s', 'e', 'start', 'end'}

Convention for converting period to timestamp; start of period vs. end

Returns **ts** : Series with DatetimeIndex

to_xarray()
Return an xarray object from the pandas object.

Returns a DataArray for a Series

a Dataset for a DataFrame

a DataArray for higher dims

Notes

See the xarray docs

Examples

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                       'B' : ['foo', 'bar', 'foo'],
                       'C' : np.arange(4.,7)})
>>> df
   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],
        [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'
```

tolist()
Convert Series to a nested list

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.

Parameters **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)

Returns **transformed** : NDFrame

See also:

`pandas.NDFrame.aggregate`, `pandas.NDFrame.apply`

Examples

```
>>> 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
```

transpose(**args*, ***kwargs*)
return the transpose, which is by definition self

truediv(*other*, *level=None*, *fill_value=None*, *axis=0*)
Floating division of series and other, element-wise (binary operator *truediv*).

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

Parameters **other** : Series or scalar value

fill_value : None or float value, default None (NaN)

Fill missing (NaN) values with this value. If both Series are missing, the result will be missing

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns **result** : Series

See also:

`Series.rtruediv`

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)
Truncates a sorted NDFrame before and/or after some particular index value. If the axis contains only datetime values, before/after parameters are converted to datetime values.

Parameters **before** : date

Truncate before index value

after : date

Truncate after index value

axis : the truncation axis, defaults to the stat axis

copy : boolean, default is True,

return a copy of the truncated section

Returns truncated : type of caller

tshift (*periods=1*, *freq=None*, *axis=0*)
Shift the time index, using the index's frequency if available.

Parameters 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

Returns shifted : NDFrame

Notes

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

tz_convert (*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

Parameters 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

Raises TypeError

If the axis is tz-naive.

tz_localize (**args*, ***kwargs*)
Localize tz-naive TimeSeries to target time zone.

Parameters 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

infer_dst : boolean, default False (DEPRECATED)

Attempt to infer fall dst-transition hours based on order

Raises **TypeError**

If the TimeSeries is tz-aware and tz is not None.

unique()

Return unique values in the object. Uniques are returned in order of appearance, this does NOT sort. Hash table-based unique.

Parameters **values** : 1d array-like

Returns unique values.

- If the input is an Index, the return is an Index
- If the input is a Categorical dtype, the return is a Categorical
- If the input is a Series/ndarray, the return will be an ndarray

See also:

unique, `Index.unique`, `Series.unique`

unstack(*level=-1*, *fill_value=None*)

Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame. The level involved will automatically get sorted.

Parameters **level** : int, string, or list of these, default last level

Level(s) to unstack, can pass level name

fill_value : replace NaN with this value if the unstack produces

missing values

Returns **unstacked** : DataFrame

Examples

```
>>> s = pd.Series([1, 2, 3, 4],
...     index=pd.MultiIndex.from_product([['one', 'two'], ['a', 'b']]))
>>> s
one  a    1
     b    2
two  a    3
     b    4
dtype: int64
```

```
>>> s.unstack(level=-1)
     a  b
one  1  2
two  3  4
```

```
>>> s.unstack(level=0)
   one  two
a    1    3
b    2    4
```

update(*other*)
Modify Series in place using non-NA values from passed Series. Aligns on index

Parameters **other** : Series

valid(*inplace=False*, ***kwargs*)

value_counts(*normalize=False*, *sort=True*, *ascending=False*, *bins=None*, *dropna=True*)
Returns object containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

Parameters **normalize** : boolean, default False

If True then the object returned will contain the relative frequencies of the unique values.

sort : boolean, default True

Sort by values

ascending : boolean, default False

Sort in ascending order

bins : integer, optional

Rather than count values, group them into half-open bins, a convenience for pd.cut, only works with numeric data

dropna : boolean, default True

Don't include counts of NaN.

Returns **counts** : Series

values
Return Series as ndarray or ndarray-like depending on the dtype

Returns **arr** : numpy.ndarray or ndarray-like

Examples

```
>>> pd.Series([1, 2, 3]).values
array([1, 2, 3])
```

```
>>> pd.Series(list('aabc')).values
array(['a', 'a', 'b', 'c'], dtype=object)
```

```
>>> pd.Series(list('aabc')).astype('category').values
[a, a, b, c]
Categories (3, object): [a, b, c]
```

Timezone aware datetime data is converted to UTC:

```
>>> pd.Series(pd.date_range('20130101', periods=3,
...                         tz='US/Eastern')).values
array(['2013-01-01T05:00:00.000000000',
       '2013-01-02T05:00:00.000000000',
       '2013-01-03T05:00:00.000000000'], dtype='datetime64[ns]')
```

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

Parameters **axis** : {index (0)}

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 scalar

ddof : int, default 1

degrees of freedom

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.

Returns **var** : scalar or Series (if level specified)

view(*dtype=None*)

where(*cond*, *other=nan*, *inplace=False*, *axis=None*, *level=None*, *try_cast=False*, *raise_on_error=True*)

Return an object of same shape as self and whose corresponding entries are from self where cond is True and otherwise are from other.

Parameters **cond** : boolean NDFrame, array-like, or callable

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

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

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)

Returns **wh** : same type as caller

See also:

`DataFrame.mask()`

Notes

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.

Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
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
```

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).

Parameters **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.

Returns **xs** : Series or DataFrame

Notes

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

Examples

```
>>> 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
second
three   5  3  5  3
```

oddt.pandas.**read_csv**(**args*, ***kwargs*)

TODO: Support Chunks

oddt.pandas.**read_mol2**(*filepath_or_buffer=None*, *usecols=None*, *molecule_column='mol'*, *molecule_name_column='mol_name'*, *smiles_column=None*, *skip_bad_mols=False*, *chunksize=None*, ***kwargs*)

Read Mol2 multi molecular file to ChemDataFrame. UCSF Dock 6 comments style is supported, i.e. *#### var_name: value* before molecular block.

New in version 0.3.

Parameters **filepath_or_buffer** : string or None

File path

usecols [list or None, optional (default=None)] A list of columns to read from file. If None then all available fields are read.

molecule_column [string or None, optional (default='mol')] Name of molecule column. If None the molecules will be skipped and the reading will be speed up significantly.

molecule_name_column [string or None, optional (default='mol_name')] Column name which will contain molecules' title/name. Column is skipped when set to None.

smiles_column [string or None, optional (default=None)] Column name containg molecules' SMILES, by default it is disabled.

skip_bad_mols [bool, optional (default=False)] Switch to skip empty (bad) molecules. Useful for RDKit, which Returns None if molecule can not sanitize.

chunksize [int or None, optional (default=None)] Size of chunk to return. If set to None whole set is returned.

Returns result :

A *ChemDataFrame* containg all molecules if *chunksize* is None or genrerator of *ChemDataFrame* with *chunksize* molecules.

oddt.pandas.**read_sdf**(*filepath_or_buffer=None*, *usecols=None*, *molecule_column='mol'*, *molecule_name_column='mol_name'*, *smiles_column=None*, *skip_bad_mols=False*, *chunksize=None*, ***kwargs*)

Read SDF/MDL multi molecular file to ChemDataFrame

New in version 0.3.

Parameters **filepath_or_buffer** : string or None

File path

usecols [list or None, optional (default=None)] A list of columns to read from file. If None then all available fields are read.

molecule_column [string or None, optional (default='mol')] Name of molecule column. If None the molecules will be skipped and the reading will be speed up significantly.

molecule_name_column [string or None, optional (default='mol_name')] Column name which will contain molecules' title/name. Column is skipped when set to None.

smiles_column [string or None, optional (default=None)] Column name containg molecules' SMILES, by default it is disabled.

skip_bad_mols [bool, optional (default=False)] Switch to skip empty (bad) molecules. Useful for RDKit, which Returns None if molecule can not sanitize.

chunksize [int or None, optional (default=None)] Size of chunk to return. If set to None whole set is returned.

Returns result :

A *ChemDataFrame* containg all molecules if *chunksize* is None or genrerator of *ChemDataFrame* with *chunksize* molecules.

5.1.8 oddt.shape module

oddt.shape.**common_usr**(*molecule*, *ctd=None*, *cst=None*, *fct=None*, *ftf=None*, *atoms_type=None*)

Function used in USR and USRCAT function

Parameters **molecule** : oddt.toolkit.Molecule

Molecule to compute USR shape descriptor

ctd : numpy array or None (default = None)

Coordinates of the molecular centroid If 'None', the point is calculated

cst : numpy array or None (default = None)

Coordinates of the closest atom to the molecular centroid If 'None', the point is calculated

fct : numpy array or None (default = None)

Coordinates of the farthest atom to the molecular centroid If 'None', the point is calculated

ftf : numpy array or None (default = None)

Coordinates of the farthest atom to the farthest atom to the molecular centroid If 'None', the point is calculated

atoms_type : str or None (default None)

Type of atoms to be selected from atom_dict If 'None', all atoms are used to calculate shape descriptor

Returns **shape_descriptor** : numpy array, shape = (12)

Array describing shape of molecule

oddt.shape.**electroshape**(*mol*)

Computes shape descriptor based on Armstrong, M. S. et al. ElectroShape: fast molecular similarity calculations incorporating shape, chirality and electrostatics. J Comput Aided Mol Des 24, 789-801 (2010). http://dx.doi.org/doi:10.1007/s10822-010-9374-0

Aside from spatial coordinates, atoms' charges are also used as the fourth dimension to describe shape of the molecule.

Parameters **mol** : oddt.toolkit.Molecule

Returns **shape_descriptor** : numpy array, shape = (15)

Array describing shape of molecule

oddt.shape.**usr**(*molecule*)

Computes USR shape descriptor based on Ballester PJ, Richards WG (2007). Ultrafast shape recognition to search compound databases for similar molecular shapes. Journal of computational chemistry, 28(10):1711-23. http://dx.doi.org/10.1002/jcc.20681

Parameters **molecule** : oddt.toolkit.Molecule

Molecule to compute USR shape descriptor

Returns **shape_descriptor** : numpy array, shape = (12)

Array describing shape of molecule

oddt.shape.**usr_cat**(*molecule*)

Computes USRCAT shape descriptor based on Adrian M Schreyer, Tom Blundell (2012). USRCAT: real-time ultrafast shape recognition with pharmacophoric constraints. Journal of Cheminformatics, 2012 4:27. http://dx.doi.org/10.1186/1758-2946-4-27

Parameters **molecule** : oddt.toolkit.Molecule

Molecule to compute USRCAT shape descriptor

Returns **shape_descriptor** : numpy array, shape = (60)

Array describing shape of molecule

oddt.shape.**usr_similarity**(*mol1_shape*, *mol2_shape*, *ow=1.0*, *hw=1.0*, *rw=1.0*, *aw=1.0*, *dw=1.0*)

Computes similarity between molecules

Parameters **mol1_shape** : numpy array

USR shape descriptor

mol2_shape : numpy array

USR shape descriptor

ow : float (default = 1.)

Scaling factor for all atoms Only used for USRCAT, ignored for other types

hw : float (default = 1.)

Scaling factor for hydrophobic atoms Only used for USRCAT, ignored for other types

rw : float (default = 1.)

Scaling factor for aromatic atoms Only used for USRCAT, ignored for other types

aw : float (default = 1.)

Scaling factor for acceptors Only used for USRCAT, ignored for other types

dw : float (default = 1.)

Scaling factor for donors Only used for USRCAT, ignored for other types

Returns **similarity** : float from 0 to 1

Similarity between shapes of molecules, 1 indicates identical molecules

5.1.9 oddt.spatial module

Spatial functions included in ODDT Mainly used by other modules, but can be accessed directly.

oddt.spatial.**angle**(*p1*, *p2*, *p3*)

Returns an angle from a series of 3 points (point #2 is centroid). Angle is returned in degrees.

Parameters **p1,p2,p3** : numpy arrays, shape = [n_points, n_dimensions]

Triplets of points in n-dimensional space, aligned in rows.

Returns **angles** : numpy array, shape = [n_points]

Series of angles in degrees

oddt.spatial.**angle_2v**(*v1*, *v2*)

Returns an angle between two vecors.Angle is returned in degrees.

Parameters **v1,v2** : numpy arrays, shape = [n_vectors, n_dimensions]

Pairs of vectors in n-dimensional space, aligned in rows.

Returns **angles** : numpy array, shape = [n_vectors]

Series of angles in degrees

oddt.spatial.**dihedral**(*p1*, *p2*, *p3*, *p4*)

Returns an dihedral angle from a series of 4 points. Dihedral is returned in degrees. Function distingishes clockwise and antyclockwise dihedrals.

Parameters **p1, p2, p3, p4** : numpy arrays, shape = [n_points, n_dimensions]

Quadruplets of points in n-dimensional space, aligned in rows.

Returns **angles** : numpy array, shape = [n_points]

Series of angles in degrees

oddt.spatial.**distance**(*x*, *y*)

Computes distance between each pair of points from x and y.

Parameters **x** : numpy arrays, shape = [n_x, 3]

Array of poinds in 3D

y : numpy arrays, shape = [n_y, 3]

Array of poinds in 3D

Returns **dist_matrix** : numpy arrays, shape = [n_x, n_y]

Distance matrix

oddt.spatial.**rmsd**(*ref*, *mol*, *ignore_h=True*, *method=None*, *normalize=False*)

Computes root mean square deviation (RMSD) between two molecules (including or excluding Hydrogens). No symmetry checks are performed.

Parameters **ref** : oddt.toolkit.Molecule object

Reference molecule for the RMSD calculation

mol : oddt.toolkit.Molecule object

Query molecule for RMSD calculation

ignore_h : bool (default=False)

Flag indicating to ignore Hydrogen atoms while performing RMSD calculation

method : str (default=None)

The method to be used for atom asignment between ref and mol. None means that direct matching is applied, which is the default behavior. Available methods:

- canonize - match heavy atoms using OB canonical ordering (it forces ignoring H's)
- hungarian - minimize RMSD using Hungarian algorithm

normalize : bool (default=False)

Normalize RMSD by square root of rot. bonds

Returns **rmsd** : float

RMSD between two molecules

oddt.spatial.**rotate**(*coords*, *alpha*, *beta*, *gamma*)

Rotate coords by cerain angle in X, Y, Z. Angles are specified in radians.

Parameters **coords** : numpy arrays, shape = [n_points, 3]

Coordinates in 3-dimensional space.

alpha, beta, gamma: float

Angles to rotate the coordinates along X, Y and Z axis. Angles are specified in radians.

Returns **new_coords** : numpy arrays, shape = [n_points, 3]

Rorated coordinates in 3-dimensional space.

5.1.10 oddt.virtualscreening module

ODDT pipeline framework for virtual screening

class oddt.virtualscreening.**virtualscreening**(*n_cpu=-1*, *verbose=False*)

Virtual Screening pipeline stack

Parameters **n_cpu: int (default=-1)**

The number of parallel procesors to use

verbose: bool (default=False) Verbosity flag for some methods

Methods

apply_filter(expression[, soft_fail])	Filtering method, can use raw expressions (strings to be evaled in if statement, can use oddt.toolkit.Molecule methods, eg.
dock(engine, protein, *args, **kwargs)	Docking procedure.
fetch()	
load_ligands(fmt, ligands_file, *args, **kwargs)	Loads file with ligands.
score(function[, protein])	Scoring procedure.
similarity(method, query[, cutoff, protein])	Similarity filter.
write(fmt, filename[, csv_filename])	Outputs molecules to a file
write_csv(csv_filename[, fields, keep_pipe])	Outputs molecules to a csv file

apply_filter (*expression*, *soft_fail=0*)
Filtering method, can use raw expressions (strings to be evaled in if statement, can use oddt.toolkit.Molecule methods, eg. 'mol.molwt < 500') Currently supported presets:

- Lipinski Rule of 5 ('ro5' or 'l5')
- Fragment Rule of 3 ('ro3')
- PAINS filter ('pains')

Parameters **expression: string or list of strings**

Expresion(s) to be used while filtering.

soft_fail: int (default=0) The number of faulures molecule can have to pass filter, aka. soft-fails.

dock (*engine*, *protein*, **args*, ***kwargs*)
Docking procedure.

Parameters **engine: string**

Which docking engine to use.

Returns None

fetch ()

load_ligands (*fmt*, *ligands_file*, **args*, ***kwargs*)
Loads file with ligands.

Parameters **file_type: string**

Type of molecular file

ligands_file: string Path to a file, which is loaded to pipeline

score (*function*, *protein=None*, **args*, ***kwargs*)
Scoring procedure.

Parameters **function: string**

Which scoring function to use.

protein: oddt.toolkit.Molecule Default protein to use as reference

similarity (*method*, *query*, *cutoff=0.9*, *protein=None*)

Similarity filter. Supported structural methods:

- ift: interaction fingerprints
- sift: simple interaction fingerprints
- usr: Ultrafast Shape recognition
- usr_cat: Ultrafast Shape recognition, Credo Atom Types
- electroshape: Electroshape, an USR method including partial charges

Parameters **method: string, one of ['ift', 'sift', 'usr', 'usr_cat', 'electroshape']**

Similarity method used to compare molecules

query: oddt.toolkit.Molecule or list of oddt.toolkit.Molecule Query molecules to compare the pipeline to.

cutoff: float Similarity cutoff for filtering molecules. Any similarity lower than it will be filtered out.

protein: oddt.toolkit.Molecule (default = None) Protein for underling method. By default it's empty, but sturctural fingerprints need one.

write (*fmt*, *filename*, *csv_filename=None*, ***kwargs*)
Outputs molecules to a file

Parameters **file_type: string**

Type of molecular file

ligands_file: string Path to a output file

csv_filename: string Optional path to a CSV file

write_csv (*csv_filename*, *fields=None*, *keep_pipe=False*, ***kwargs*)
Outputs molecules to a csv file

Parameters **csv_filename: string**

Optional path to a CSV file

fields: list (default None) List of fields to save in CSV file

keep_pipe: bool (default=False) If set to True, the ligand pipe is sustained.

5.1.11 Module contents

Open Drug Discovery Toolkit

Universal and easy to use resource for various drug discovery tasks, ie docking, virutal screening, rescoring.

toolkit [module,] Toolkits backend module, currenlty OpenBabel [ob] and RDKit [rdk]. This setting is toolkit-wide, and sets given toolkit as default

CHAPTER 6

References

To be announced.

CHAPTER 7

Documentation Indices and tables

- genindex
- modindex
- search

Bibliography

[R1] Wikipedia entry for the Receiver operating characteristic

Python Module Index

o

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