
formulas Documentation

Release 1.2.7

Vincenzo Arcidiacono

Nov 15, 2023

TABLE OF CONTENTS

1	What is formulas?	3
2	Installation	5
2.1	Install extras	5
2.2	Development version	5
2.2.1	What is formulas?	5
2.2.2	Installation	6
2.2.2.1	Install extras	6
2.2.2.2	Development version	6
2.2.3	Basic Examples	6
2.2.3.1	Parsing formula	6
2.2.3.2	Excel workbook	8
2.2.3.3	Custom functions	13
2.2.4	Next moves	13
2.2.5	Contributing to formulas	13
2.2.5.1	Clone the repository	13
2.2.5.2	How to implement a new function	14
2.2.5.3	How to open a pull request	14
2.2.6	Donate	15
2.2.7	API Reference	15
2.2.7.1	parser	16
2.2.7.2	builder	17
2.2.7.3	errors	19
2.2.7.4	tokens	20
2.2.7.5	functions	60
2.2.7.6	ranges	301
2.2.7.7	cell	303
2.2.7.8	excel	308
2.2.8	Changelog	326
2.2.8.1	v1.2.6 (2023-11-15)	326
2.2.8.2	v1.2.6 (2022-12-13)	327
2.2.8.3	v1.2.5 (2022-11-07)	327
2.2.8.4	v1.2.4 (2022-07-02)	327
2.2.8.5	v1.2.3 (2022-05-10)	328
2.2.8.6	v1.2.2 (2022-01-22)	328
2.2.8.7	v1.2.1 (2022-01-21)	328
2.2.8.8	v1.2.0 (2021-12-23)	329
2.2.8.9	v1.1.1 (2021-10-13)	329
2.2.8.10	v1.1.0 (2021-02-16)	330
2.2.8.11	v1.0.0 (2020-03-12)	331

2.2.8.12	v0.4.0 (2019-08-31)	332
2.2.8.13	v0.3.0 (2019-04-24)	332
2.2.8.14	v0.2.0 (2018-12-11)	333
2.2.8.15	v0.1.4 (2018-10-19)	333
2.2.8.16	v0.1.3 (2018-10-09)	334
2.2.8.17	v0.1.2 (2018-09-12)	334
2.2.8.18	v0.1.1 (2018-09-11)	335
2.2.8.19	v0.1.0 (2018-07-20)	335
2.2.8.20	v0.0.10 (2018-06-05)	335
2.2.8.21	v0.0.9 (2018-05-28)	336
2.2.8.22	v0.0.8 (2018-05-23)	336
2.2.8.23	v0.0.7 (2017-07-20)	337
2.2.8.24	v0.0.6 (2017-05-31)	338
2.2.8.25	v0.0.5 (2017-05-04)	338
2.2.8.26	v0.0.4 (2017-02-10)	339
2.2.8.27	v0.0.3 (2017-02-09)	339
2.2.8.28	v0.0.2 (2017-02-08)	339
3	Indices and tables	341
	Python Module Index	343
	Index	345

2023-11-15 01:00:00

<https://github.com/vinci1it2000/formulas>

<https://pypi.org/project/formulas/>

<http://formulas.readthedocs.io/>

<https://github.com/vinci1it2000/formulas/wiki/>

<http://github.com/vinci1it2000/formulas/releases/>

<https://donorbox.org/formulas>

excel, formulas, interpreter, compiler, dispatch

- Vincenzo Arcidiacono <vinci1it2000@gmail.com>

EUPL 1.1+

WHAT IS FORMULAS?

formulas implements an interpreter for Excel formulas, which parses and compile Excel formulas expressions.

Moreover, it compiles Excel workbooks to python and executes without using the Excel COM server. Hence, **Excel is not needed**.

INSTALLATION

To install it use (with root privileges):

```
$ pip install formulas
```

Or download the last git version and use (with root privileges):

```
$ python setup.py install
```

2.1 Install extras

Some additional functionality is enabled installing the following extras:

- excel: enables to compile Excel workbooks to python and execute using: *ExcelModel*.
- plot: enables to plot the formula ast and the Excel model.

To install formulas and all extras, do:

```
$ pip install formulas[all]
```

2.2 Development version

To help with the testing and the development of *formulas*, you can install the development version:

```
$ pip install https://github.com/vincilit2000/formulas/archive/dev.zip
```

2.2.1 What is formulas?

formulas implements an interpreter for Excel formulas, which parses and compile Excel formulas expressions.

Moreover, it compiles Excel workbooks to python and executes without using the Excel COM server. Hence, **Excel is not needed**.

2.2.2 Installation

To install it use (with root privileges):

```
$ pip install formulas
```

Or download the last git version and use (with root privileges):

```
$ python setup.py install
```

2.2.2.1 Install extras

Some additional functionality is enabled installing the following extras:

- excel: enables to compile Excel workbooks to python and execute using: [ExcelModel](#).
- plot: enables to plot the formula ast and the Excel model.

To install formulas and all extras, do:

```
$ pip install formulas[all]
```

2.2.2.2 Development version

To help with the testing and the development of *formulas*, you can install the development version:

```
$ pip install https://github.com/vinci1it2000/formulas/archive/dev.zip
```

2.2.3 Basic Examples

The following sections will show how to:

- parse a Excel formulas;
- load, compile, and execute a Excel workbook;
- extract a sub-model from a Excel workbook;
- add a custom function.

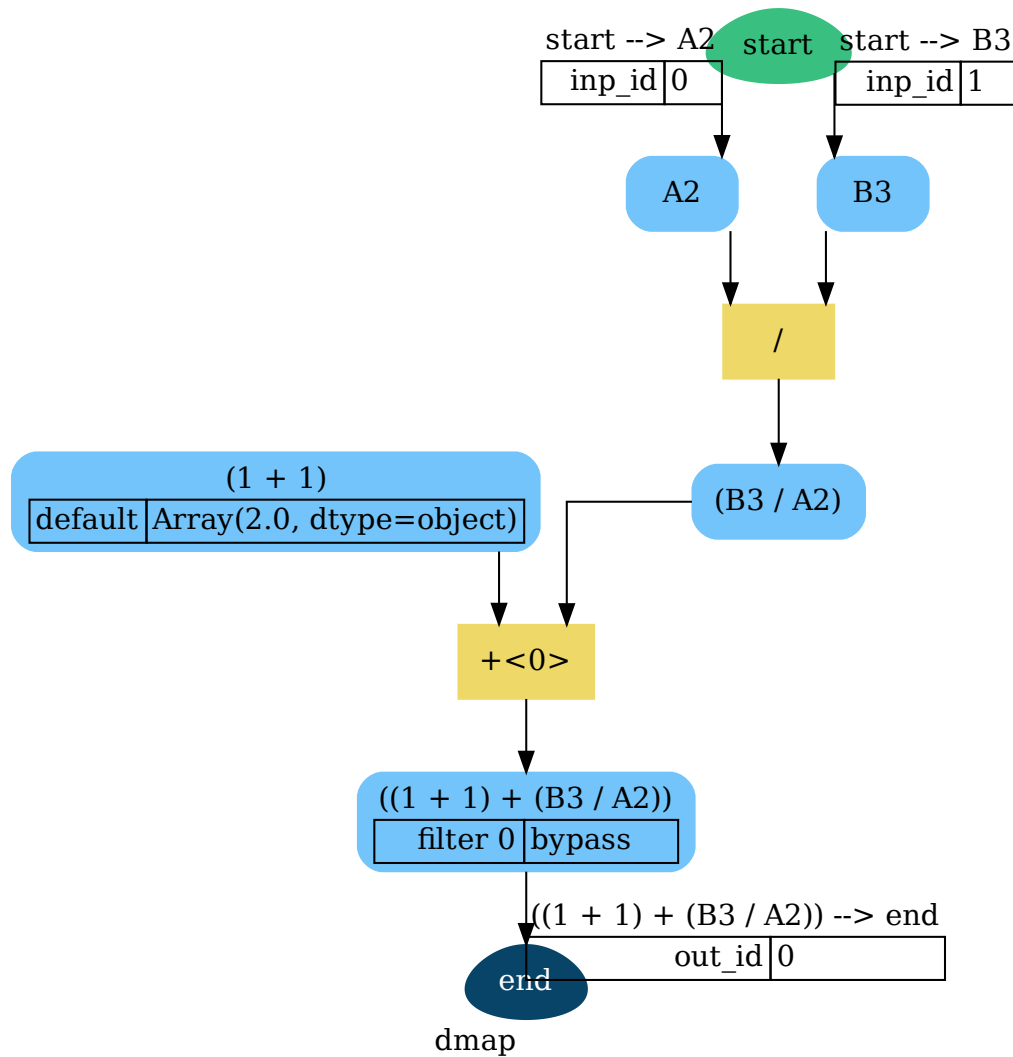
2.2.3.1 Parsing formula

An example how to parse and execute an Excel formula is the following:

```
>>> import formulas
>>> func = formulas.Parser().ast('=(1 + 1) + B3 / A2')[1].compile()
```

To visualize formula model and get the input order you can do the following:

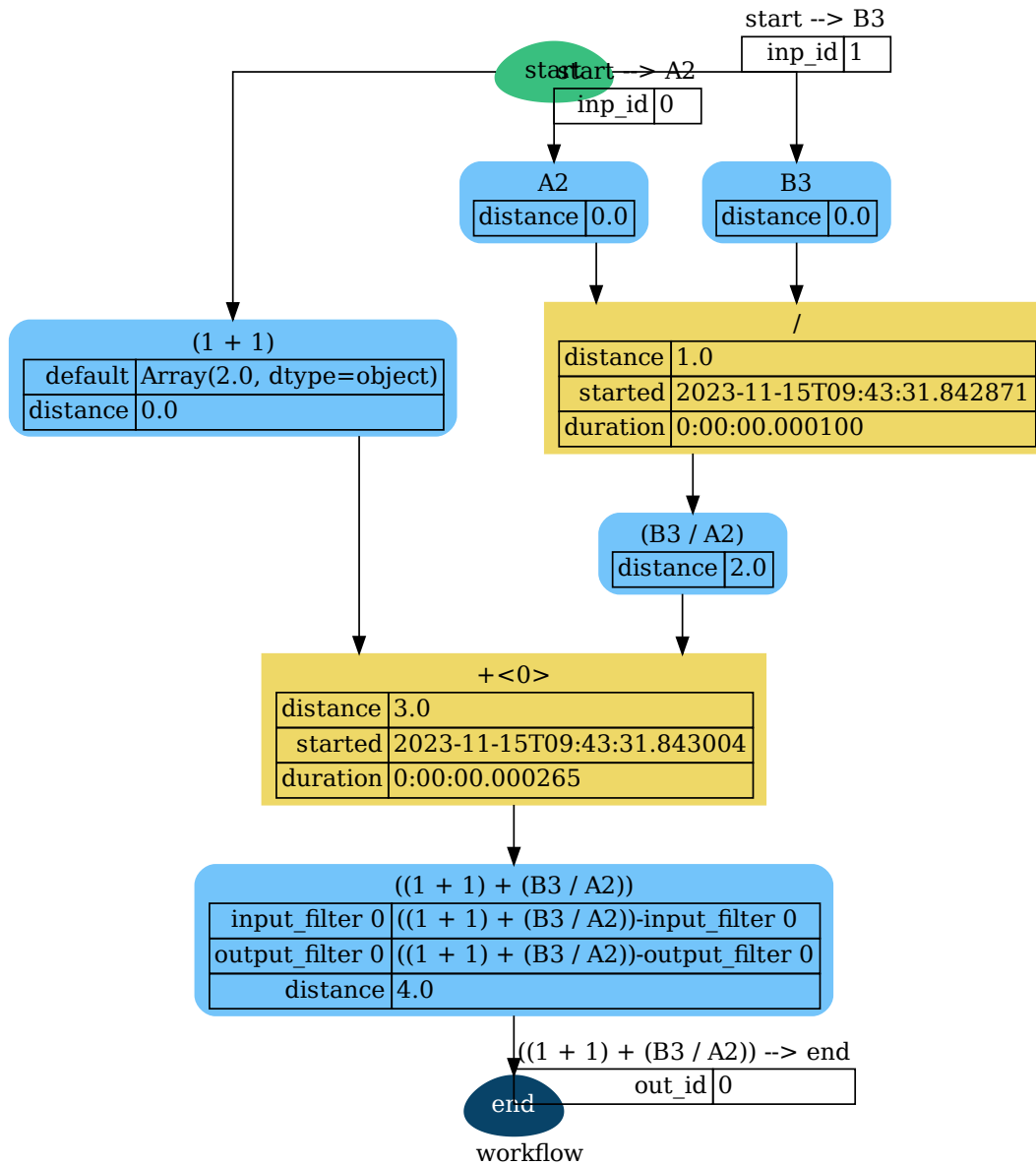
```
>>> list(func.inputs)
['A2', 'B3']
>>> func.plot(view=False) # Set view=True to plot in the default browser.
SiteMap([(=(1 + 1) + (B3 / A2)), SiteMap())])
```



Finally to execute the formula and plot the workflow:

```

>>> func(1, 5)
Array(7.0, dtype=object)
>>> func.plot(workflow=True, view=False) # Set view=True to plot in the default browser.
SiteMap([(=(1 + 1) + (B3 / A2)), SiteMap())])
  
```



2.2.3.2 Excel workbook

An example how to load, calculate, and write an Excel workbook is the following:

```

>>> import formulas
>>> fpath, dir_output = 'excel.xlsx', 'output'
>>> xl_model = formulas.ExcelModel().loads(fpath).finish()
>>> xl_model.calculate()
Solution(...)
>>> xl_model.write(dirpath=dir_output)
  
```

(continues on next page)

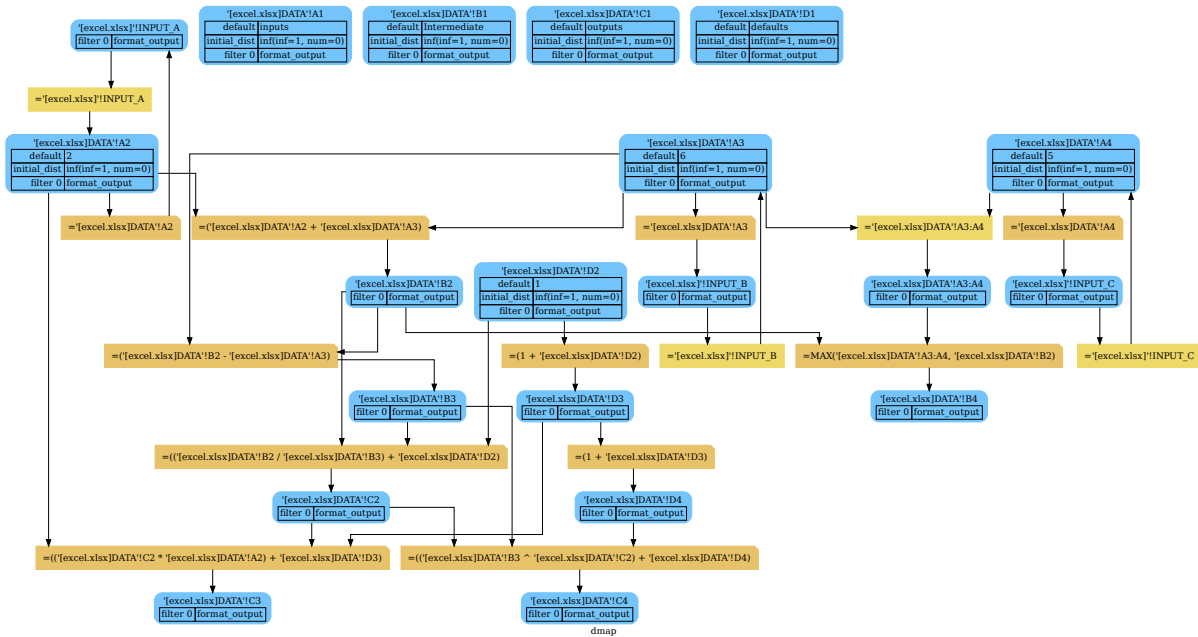
(continued from previous page)

```
{'EXCEL.XLSX': {Book: <openpyxl.workbook.workbook.Workbook ...>}}
```

Tip: If you have or could have **circular references**, add `circular=True` to `finish` method.

To plot the dependency graph that depict relationships between Excel cells:

```
>>> dsp = xl_model.dsp
>>> dsp.plot(view=False) # Set view=True to plot in the default browser.
SiteMap([(ExcelModel, SiteMap(...))])
```



To overwrite the default inputs that are defined by the excel file or to impose some value to a specific cell:

```
>>> xl_model.calculate(
...     inputs={
...         "'[excel.xlsx]!INPUT_A': 3, # To overwrite the default value.
...         "'[excel.xlsx]DATA!B3': 1 # To impose a value to B3 cell.
...     },
...     outputs=[
...         "'[excel.xlsx]DATA!C2", "'[excel.xlsx]DATA!C4"
...     ] # To define the outputs that you want to calculate.
... )
Solution([( "'[excel.xlsx]!INPUT_A", <Ranges>(' [excel.xlsx]DATA!A2)=[ [3]] ),
( "'[excel.xlsx]DATA!B3", <Ranges>(' [excel.xlsx]DATA!B3)=[ [1]] ),
( "'[excel.xlsx]DATA!A2", <Ranges>(' [excel.xlsx]DATA!A2)=[ [3]] ),
( "'[excel.xlsx]DATA!A3", <Ranges>(' [excel.xlsx]DATA!A3)=[ [6]] ),
( "'[excel.xlsx]DATA!D2", <Ranges>(' [excel.xlsx]DATA!D2)=[ [1]] ),
( "'[excel.xlsx]!INPUT_B", <Ranges>(' [excel.xlsx]DATA!A3)=[ [6]] ),
( "'[excel.xlsx]DATA!B2", <Ranges>(' [excel.xlsx]DATA!B2)=[ [9.0]] ),
( "'[excel.xlsx]DATA!D3", <Ranges>(' [excel.xlsx]DATA!D3)=[ [2.0]] ),
```

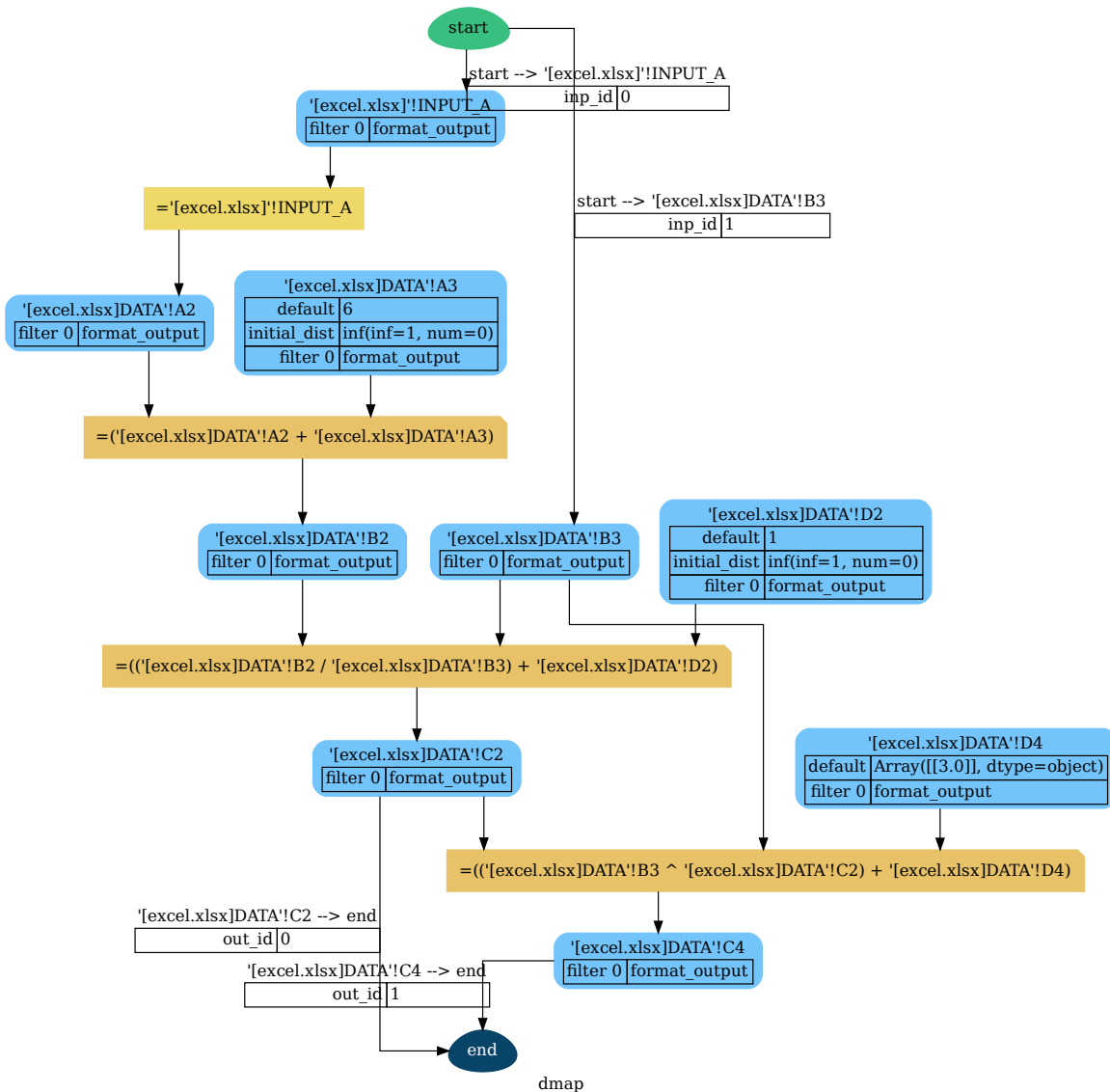
(continues on next page)

(continued from previous page)

```
("'[excel.xlsx]DATA'!C2", <Ranges>('[excel.xlsx]DATA'!C2)=[[10.0]]),  
("[excel.xlsx]DATA'!D4", <Ranges>('[excel.xlsx]DATA'!D4)=[[3.0]]),  
("[excel.xlsx]DATA'!C4", <Ranges>('[excel.xlsx]DATA'!C4)=[[4.0]]))
```

To build a single function out of an excel model with fixed inputs and outputs, you can use the *compile* method of the *ExcelModel* that returns a *DispatchPipe*. This is a function where the inputs and outputs are defined by the data node ids (i.e., cell references).

```
>>> func = xl_model.compile(  
...     inputs=[  
...         "'[excel.xlsx]!INPUT_A", # First argument of the function.  
...         "'[excel.xlsx]DATA'!B3"  # Second argument of the function.  
...     ], # To define function inputs.  
...     outputs=[  
...         "'[excel.xlsx]DATA'!C2", "'[excel.xlsx]DATA'!C4"  
...     ] # To define function outputs.  
... )  
>>> func  
<schedula.utils.dsp.DispatchPipe object at ...>  
>>> [v.value[0, 0] for v in func(3, 1)] # To retrieve the data.  
[10.0, 4.0]  
>>> func.plot(view=False) # Set view=True to plot in the default browser.  
SiteMap([(ExcelModel, SiteMap(...))])
```



Alternatively, to load a partial excel model from the output cells, you can use the *from_ranges* method of the *ExcelModel*:

```

>>> x1 = formulas.ExcelModel().from_ranges(
...     "'[%s]DATA!C2:D2" % fpath, # Output range.
...     "'[%s]DATA!B4" % fpath, # Output cell.
... )
>>> dsp = x1.dsp
>>> sorted(dsp.data_nodes)
["'[excel.xlsx]!INPUT_A",
 "'[excel.xlsx]!INPUT_B",
 "'[excel.xlsx]!INPUT_C",
 "'[excel.xlsx]DATA!A2",
 "'[excel.xlsx]DATA!A3",

```

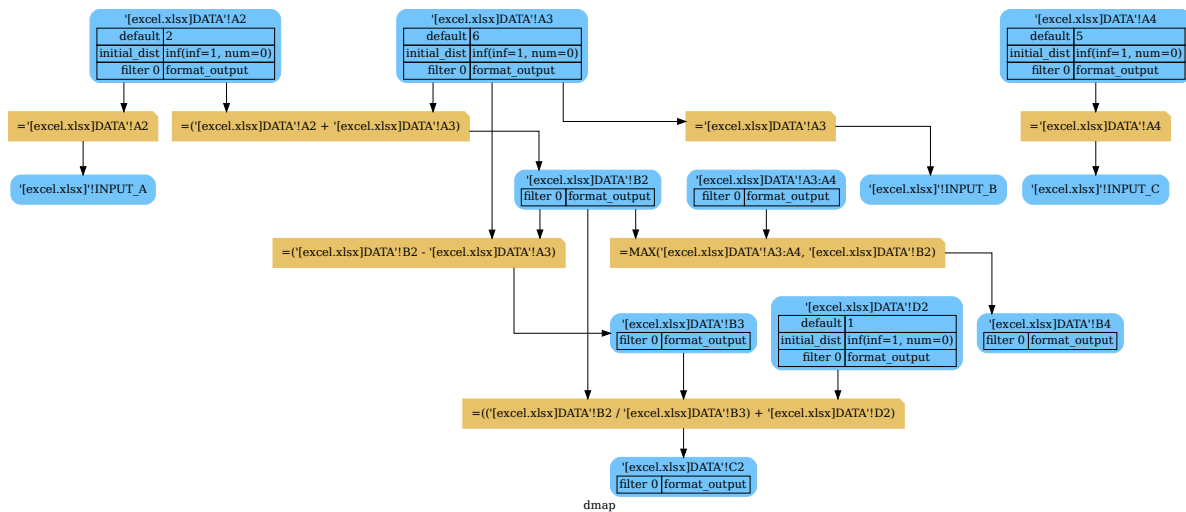
(continues on next page)

(continued from previous page)

```

" '[excel.xlsx]DATA'!A3:A4",
" '[excel.xlsx]DATA'!A4",
" '[excel.xlsx]DATA'!B2",
" '[excel.xlsx]DATA'!B3",
" '[excel.xlsx]DATA'!B4",
" '[excel.xlsx]DATA'!C2",
" '[excel.xlsx]DATA'!D2"]

```



JSON export/import

The *ExcelModel* can be exported/imported to/from a readable JSON format. The reason of this functionality is to have format that can be easily maintained (e.g. using version control programs like *git*). Follows an example on how to export/import to/from JSON an *ExcelModel*:

```

>>> import json
>>> xl_dict = xl_model.to_dict() # To JSON-able dict.
>>> xl_dict # Exported format.
{
  '[excel.xlsx]DATA'!A1": "inputs",
  '[excel.xlsx]DATA'!B1": "Intermediate",
  '[excel.xlsx]DATA'!C1": "outputs",
  '[excel.xlsx]DATA'!D1": "defaults",
  '[excel.xlsx]DATA'!A2": 2,
  '[excel.xlsx]DATA'!D2": 1,
  '[excel.xlsx]DATA'!A3": 6,
  '[excel.xlsx]DATA'!A4": 5,
  '[excel.xlsx]DATA'!B2": "=( '[excel.xlsx]DATA'!A2 + '[excel.xlsx]DATA'!A3)",
  '[excel.xlsx]DATA'!C2": "=((' [excel.xlsx]DATA'!B2 / '[excel.xlsx]DATA'!B3) + '[excel.
↪xlxs]DATA'!D2)",
  '[excel.xlsx]DATA'!B3": "=( '[excel.xlsx]DATA'!B2 - '[excel.xlsx]DATA'!A3)",
  '[excel.xlsx]DATA'!C3": "=((' [excel.xlsx]DATA'!C2 * '[excel.xlsx]DATA'!A2) + '[excel.
↪xlxs]DATA'!D3)",

```

(continues on next page)

(continued from previous page)

```

'''[excel.xlsx]DATA'!D3": "=(1 + '[excel.xlsx]DATA'!D2)",
'''[excel.xlsx]DATA'!B4": "=MAX('[excel.xlsx]DATA'!A3:A4, '[excel.xlsx]DATA'!B2)",
'''[excel.xlsx]DATA'!C4": "=(('[excel.xlsx]DATA'!B3 ^ '[excel.xlsx]DATA'!C2) + '[excel.
↪xlsx]DATA'!D4)",
'''[excel.xlsx]DATA'!D4": "=(1 + '[excel.xlsx]DATA'!D3)"
}
>>> xl_json = json.dumps(xl_dict, indent=True) # To JSON.
>>> xl_model = formulas.ExcelModel().from_dict(json.loads(xl_json)) # From JSON.

```

2.2.3.3 Custom functions

An example how to add a custom function to the formula parser is the following:

```

>>> import formulas
>>> FUNCTIONS = formulas.get_functions()
>>> FUNCTIONS['MYFUNC'] = lambda x, y: 1 + y + x
>>> func = formulas.Parser().ast('=MYFUNC(1, 2)')[1].compile()
>>> func()
4

```

2.2.4 Next moves

Things yet to do: implement the missing Excel formulas.

2.2.5 Contributing to formulas

If you want to contribute to **formulas** and make it better, your help is very welcome. The contribution should be sent by a *pull request*. Next sections will explain how to implement and submit a new excel function:

- clone the repository
- implement a new function/functionality
- open a pull request

2.2.5.1 Clone the repository

The first step to contribute to **formulas** is to clone the repository:

- Create a personal [fork](#) of the [formulas](#) repository on Github.
- [Clone](#) the fork on your local machine. Your remote repo on Github is called `origin`.
- [Add](#) the original repository as a remote called `upstream`, to maintain updated your fork.
- If you created your fork a while ago be sure to pull `upstream` changes into your local repository.
- Create a new branch to work on! Branch from `dev`.

2.2.5.2 How to implement a new function

Before coding, [study](#) the Excel function that you want to implement. If there is something similar implemented in **formulas**, try to get inspired by the implemented code (I mean, not reinvent the wheel) and to use **numpy**. Follow the code style of the project, including indentation. Add or change the documentation as needed. Make sure that you have implemented the **full function syntax**, including the [array syntax](#).

Test cases are very important. This library uses a data-driven testing approach. To implement a new function I recommend the [test-driven development cycle](#). Hence, when you implement a new function, you should write new test cases in `test_cell/TestCell.test_output` suite to execute in the *cycle loop*. When you think that the code is ready, add new raw test in `test/test_files/test.xlsx` (please follow the standard used for other functions) and run the `test_excel/TestExcelModel.test_excel_model`. This requires more time but is needed to test the **array syntax** and to check if the Excel documentation respects the reality.

When all test cases are ok (`python setup.py test`), open a pull request.

Do do list:

- Study the excel function syntax and behaviour when used as array formula.
- Check if there is something similar implemented in formulas.
- Implement/fix your feature, comment your code.
- Write/adapt tests and run them!

Tip: Excel functions are categorized by their functionality. If you are implementing a new functionality group, add a new module in `formula/function` and in `formula.function.SUBMODULES` and a new worksheet in `test/test_files/test.xlsx` (please respect the format).

Note: A pull request without new test case will not be taken into consideration.

2.2.5.3 How to open a pull request

Well done! Your contribution is ready to be submitted:

- Squash your commits into a single commit with git's [interactive rebase](#). Create a new branch if necessary. Always write your commit messages in the present tense. Your commit message should describe what the commit, when applied, does to the code – not what you did to the code.
- [Push](#) your branch to your fork on Github (i.e., `git push origin dev`).
- From your fork [open a pull request](#) in the correct branch. Target the project's dev branch!
- Once the *pull request* is approved and merged you can pull the changes from *upstream* to your local repo and delete your extra branch(es).

2.2.6 Donate

If you want to [support](#) the **formulas** development please donate and add your excel function preferences. The selection of the functions to be implemented is done considering the cumulative donation amount per function collected by the campaign.

Note: The cumulative donation amount per function is calculated as the example:

Function	Donator 1	Donator 2	Donator 3	TOT	Implementa- tion order
.	150€	120€	50€	.	.
SUM	50€	40€	25€	125€	1st
SIN	50€		25€	75€	3rd
TAN	50€	40€		90€	2nd
COS		40€		40€	4th

2.2.7 API Reference

The core of the library is composed from the following modules: It contains a comprehensive list of all modules and classes within formulas.

Modules:

<i>parser</i>	It provides formula parser class.
<i>builder</i>	It provides AstBuilder class.
<i>errors</i>	Defines the formulas exception.
<i>tokens</i>	It provides tokens needed to parse the Excel formulas.
<i>functions</i>	It provides functions implementations to compile the Excel functions.
<i>ranges</i>	It provides Ranges class.
<i>cell</i>	It provides Cell class.
<i>excel</i>	It provides Excel model class.

2.2.7.1 parser

It provides formula parser class.

Classes

<i>Parser</i>

Parser

class Parser

Methods

<code>__init__</code>

<code>ast</code>

<code>is_formula</code>

`__init__`

Parser.`__init__`()

ast

Parser.**ast**(*expression*, *context=None*)

is_formula

Parser.**is_formula**(*value*)

`__init__`()

Attributes

filters
formula_check

filters

```
Parser.filters = [<class 'formulas.tokens.operand.Error'>, <class
'formulas.tokens.operand.String'>, <class 'formulas.tokens.operand.Number'>, <class
'formulas.tokens.operand.Range'>, <class 'formulas.tokens.operator.OperatorToken'>,
<class 'formulas.tokens.operator.Separator'>, <class
'formulas.tokens.function.Function'>, <class 'formulas.tokens.function.Array'>,
<class 'formulas.tokens.parenthesis.Parenthesis'>, <class
'formulas.tokens.operator.Intersect'>]
```

formula_check

```
Parser.formula_check = regex.Regex('\n
(?P<array>^\\s*{\\s*=\\s*(?P<name>\\S.*)\\s*}\\s*$)\n |\n
(?P<value>^\\s*=\\s*(?P<name>\\S.*))\n ', flags=regex.S | regex.I | regex.X |
regex.V0)
```

2.2.7.2 builder

It provides AstBuilder class.

Classes

AstBuilder

AstBuilder

```
class AstBuilder(dsp=None, nodes=None, match=None)
```

Methods

<code>__init__</code>
<code>append</code>
<code>compile</code>
<code>finish</code>
<code>get_node_id</code>
<code>pop</code>

`__init__`

`AstBuilder.__init__(dsp=None, nodes=None, match=None)`

`append`

`AstBuilder.append(token)`

`compile`

`AstBuilder.compile(references=None, context=None, **inputs)`

`finish`

`AstBuilder.finish()`

`get_node_id`

`AstBuilder.get_node_id(token)`

`pop`

`AstBuilder.pop()`

`__init__(dsp=None, nodes=None, match=None)`

`compile_class`

alias of `DispatchPipe`

2.2.7.3 errors

Defines the formulas exception.

Exceptions

BaseError
BroadcastError
FormulaError
FoundError
FunctionError
InvalidRangeError
InvalidRangeName
ParenthesesError
RangeValueError
TokenError

BaseError

```
exception BaseError(*args)
```

BroadcastError

```
exception BroadcastError(*args)
```

FormulaError

```
exception FormulaError(*args)
```

FoundError

exception `FoundError(*args, err=None, **kwargs)`

FunctionError

exception `FunctionError(*args)`

InvalidRangeError

exception `InvalidRangeError(*args)`

InvalidRangeName

exception `InvalidRangeName`

ParenthesesError

exception `ParenthesesError(*args)`

RangeValueError

exception `RangeValueError(*args)`

TokenError

exception `TokenError(*args)`

2.2.7.4 tokens

It provides tokens needed to parse the Excel formulas.

Sub-Modules:

<i>function</i>	It provides Function classes.
<i>operand</i>	It provides Operand classes.
<i>operator</i>	It provides Operator classes.
<i>parenthesis</i>	It provides Parenthesis class.

function

It provides Function classes.

Classes

Array

Function

Array

class `Array(s, context=None)`

Methods

`__init__`

`ast`

`compile`

`match`

`process`

`set_expr`

`update_input_tokens`

`__init__`

`Array.__init__(s, context=None)`

ast

`Array.ast(tokens, stack, builder, check_n=<function Array.<lambda>>)`

compile

`Array.compile()`

match

`Array.match(s)`

process

`Array.process(match, context=None)`

set_expr

`Array.set_expr(*tokens)`

update_input_tokens

`Array.update_input_tokens(*tokens)`

`__init__(s, context=None)`

Attributes

name
node_id

name

property `Array.name`

node_id

property `Array.node_id`

Function

class **Function**(*s*, *context=None*)

Methods

<code>__init__</code>
<code>ast</code>
<code>compile</code>
<code>match</code>
<code>process</code>
<code>set_expr</code>
<code>update_input_tokens</code>

`__init__`

Function.`__init__`(*s*, *context=None*)

ast

Function.`ast`(*tokens*, *stack*, *builder*, *check_n=<function Function.<lambda>>*)

compile

Function.`compile`()

match

Function.`match`(*s*)

process

Function.**process**(*match*, *context=None*)

set_expr

Function.**set_expr**(**tokens*)

update_input_tokens

Function.**update_input_tokens**(**tokens*)

__init__(*s*, *context=None*)

Attributes

name
node_id

name

property Function.**name**

node_id

property Function.**node_id**

operand

It provides Operand classes.

Functions

<i>fast_range2parts</i>
<i>fast_range2parts_v1</i>
<i>fast_range2parts_v2</i>
<i>fast_range2parts_v3</i>
<i>fast_range2parts_v4</i>
<i>fast_range2parts_v5</i>
<i>range2parts</i>

fast_range2parts

fast_range2parts(**kw)

fast_range2parts_v1

fast_range2parts_v1(r1, c1, sheet_id)

fast_range2parts_v2

fast_range2parts_v2(r1, c1, r2, c2, sheet_id)

fast_range2parts_v3

fast_range2parts_v3(r1, n1, sheet_id)

fast_range2parts_v4

fast_range2parts_v4(r1, n1, r2, n2, sheet_id)

fast_range2parts_v5

fast_range2parts_v5(*ref*, *sheet_id*)

range2parts

range2parts(*outputs*, ***inputs*)

Classes

<i>Empty</i>
<i>Error</i>
<i>Number</i>
<i>Operand</i>
<i>Range</i>
<i>String</i>
<i>XLError</i>

Empty

class **Empty**

Methods

<code>__init__</code>
<code>ast</code>
<code>compile</code>
<code>match</code>
<code>process</code>
<code>set_expr</code>
<code>update_input_tokens</code>

__init__

`Empty.__init__()`

ast

`Empty.ast(tokens, stack, builder)`

compile

`static Empty.compile()`

match

`Empty.match(s)`

process

`Empty.process(match, context=None)`

set_expr

`Empty.set_expr(*tokens)`

update_input_tokens

`Empty.update_input_tokens(*tokens)`

`__init__()`

Attributes

name

node_id

name

property `Empty.name`

node_id

property `Empty.node_id`

Error

class `Error(s, context=None)`

Methods

<code>__init__</code>
<code>ast</code>
<code>compile</code>
<code>match</code>
<code>process</code>
<code>set_expr</code>
<code>update_input_tokens</code>

`__init__`

`Error.__init__(s, context=None)`

ast

`Error.ast(tokens, stack, builder)`

compile

`Error.compile()`

match

`Error.match(s)`

process

`Error.process(match, context=None)`

set_expr

`Error.set_expr(*tokens)`

update_input_tokens

`Error.update_input_tokens(*tokens)`

`__init__(s, context=None)`

Attributes

errors
name
node_id

errors

`Error.errors = {'#DIV/0!': '#DIV/0!', '#N/A': '#N/A', '#NAME?': '#NAME?', '#NULL!': '#NULL!', '#NUM!': '#NUM!', '#REF!': '#REF!', '#VALUE!': '#VALUE!'}`

name

property `Error.name`

node_id

property `Error.node_id`

Number

class `Number(s, context=None)`

Methods

<code>__init__</code>
<code>ast</code>
<code>compile</code>
<code>match</code>
<code>process</code>
<code>set_expr</code>
<code>update_input_tokens</code>

`__init__`

`Number.__init__(s, context=None)`

ast

`Number.ast(tokens, stack, builder)`

compile

`Number.compile()`

match

`Number.match(s)`

process

`Number.process(match, context=None)`

set_expr

`Number.set_expr(*tokens)`

update_input_tokens

`Number.update_input_tokens(*tokens)`

`__init__(s, context=None)`

Attributes

name
node_id

name

property `Number.name`

node_id

property `Number.node_id`

Operand

class `Operand`(*s*, *context=None*)

Methods

`__init__``ast``match``process``set_expr``update_input_tokens`

`__init__`

`Operand.__init__(s, context=None)`

`ast`

`Operand.ast(tokens, stack, builder)`

`match`

`Operand.match(s)`

`process`

`Operand.process(match, context=None)`

`set_expr`

`Operand.set_expr(*tokens)`

update_input_tokens

Operand.**update_input_tokens**(*tokens)

__init__(s, context=None)

Attributes

name
node_id

name

property Operand.**name**

node_id

property Operand.**node_id**

Range

class Range(s, context=None)

Methods

<code>__init__</code>
ast
compile
match
process
set_expr
update_input_tokens

`__init__`

`Range.__init__(s, context=None)`

`ast`

`Range.ast(tokens, stack, builder)`

`compile`

`Range.compile()`

`match`

`Range.match(s)`

`process`

`Range.process(match, context=None)`

`set_expr`

`Range.set_expr(*tokens)`

`update_input_tokens`

`Range.update_input_tokens(*tokens)`

`__init__(s, context=None)`

Attributes

name
node_id

name

property Range.name

node_id

property Range.node_id

String

class String(*s, context=None*)

Methods

<code>__init__</code>
<code>ast</code>
<code>compile</code>
<code>match</code>
<code>process</code>
<code>set_expr</code>
<code>update_input_tokens</code>

`__init__`

String.__init__(*s, context=None*)

ast

String.ast(*tokens, stack, builder*)

compile

`String.compile()`

match

`String.match(s)`

process

`String.process(match, context=None)`

set_expr

`String.set_expr(*tokens)`

update_input_tokens

`String.update_input_tokens(*tokens)`

`__init__(s, context=None)`

Attributes

name
node_id

name

property `String.name`

node_id

property `String.node_id`

XLError

class `XLError(*args)`

Methods

<code>__init__</code>	
<code>capitalize</code>	Return a capitalized version of the string.
<code>casefold</code>	Return a version of the string suitable for caseless comparisons.
<code>center</code>	Return a centered string of length width.
<code>count</code>	Return the number of non-overlapping occurrences of substring sub in string S[start:end].
<code>encode</code>	Encode the string using the codec registered for encoding.
<code>endswith</code>	Return True if S ends with the specified suffix, False otherwise.
<code>expandtabs</code>	Return a copy where all tab characters are expanded using spaces.
<code>find</code>	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
<code>format</code>	Return a formatted version of S, using substitutions from args and kwargs.
<code>format_map</code>	Return a formatted version of S, using substitutions from mapping.
<code>index</code>	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
<code>isalnum</code>	Return True if the string is an alpha-numeric string, False otherwise.
<code>isalpha</code>	Return True if the string is an alphabetic string, False otherwise.
<code>isascii</code>	Return True if all characters in the string are ASCII, False otherwise.
<code>isdecimal</code>	Return True if the string is a decimal string, False otherwise.
<code>isdigit</code>	Return True if the string is a digit string, False otherwise.
<code>isidentifier</code>	Return True if the string is a valid Python identifier, False otherwise.
<code>islower</code>	Return True if the string is a lowercase string, False otherwise.
<code>isnumeric</code>	Return True if the string is a numeric string, False otherwise.
<code>isprintable</code>	Return True if the string is printable, False otherwise.
<code>isspace</code>	Return True if the string is a whitespace string, False otherwise.
<code>istitle</code>	Return True if the string is a title-cased string, False otherwise.
<code>isupper</code>	Return True if the string is an uppercase string, False otherwise.

continues on next page

Table 1 – continued from previous page

<code>join</code>	Concatenate any number of strings.
<code>ljust</code>	Return a left-justified string of length width.
<code>lower</code>	Return a copy of the string converted to lowercase.
<code>lstrip</code>	Return a copy of the string with leading whitespace removed.
<code>maketrans</code>	Return a translation table usable for <code>str.translate()</code> .
<code>partition</code>	Partition the string into three parts using the given separator.
<code>removeprefix</code>	Return a str with the given prefix string removed if present.
<code>removesuffix</code>	Return a str with the given suffix string removed if present.
<code>replace</code>	Return a copy with all occurrences of substring old replaced by new.
<code>rfind</code>	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
<code>rindex</code>	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
<code>rjust</code>	Return a right-justified string of length width.
<code>rpartition</code>	Partition the string into three parts using the given separator.
<code>rsplit</code>	Return a list of the substrings in the string, using sep as the separator string.
<code>rstrip</code>	Return a copy of the string with trailing whitespace removed.
<code>split</code>	Return a list of the substrings in the string, using sep as the separator string.
<code>splitlines</code>	Return a list of the lines in the string, breaking at line boundaries.
<code>startswith</code>	Return True if S starts with the specified prefix, False otherwise.
<code>strip</code>	Return a copy of the string with leading and trailing whitespace removed.
<code>swapcase</code>	Convert uppercase characters to lowercase and lowercase characters to uppercase.
<code>title</code>	Return a version of the string where each word is titlecased.
<code>translate</code>	Replace each character in the string using the given translation table.
<code>upper</code>	Return a copy of the string converted to uppercase.
<code>zfill</code>	Pad a numeric string with zeros on the left, to fill a field of the given width.

`__init__`

`XLError.__init__(*args)`

`capitalize`

`XLError.capitalize()`

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

`casefold`

`XLError.casefold()`

Return a version of the string suitable for caseless comparisons.

`center`

`XLError.center(width, fillchar=' ', /)`

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

`count`

`XLError.count(sub[, start[, end]]) → int`

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

`encode`

`XLError.encode(encoding='utf-8', errors='strict')`

Encode the string using the codec registered for encoding.

`encoding`

The encoding in which to encode the string.

`errors`

The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a `UnicodeEncodeError`. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with `codecs.register_error` that can handle `UnicodeEncodeErrors`.

endswith

`XLError.endswith(suffix[, start[, end]]) → bool`

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs

`XLError.expandtabs(tabsize=8)`

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find

`XLError.find(sub[, start[, end]]) → int`

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format

`XLError.format(*args, **kwargs) → str`

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map

`XLError.format_map(mapping) → str`

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index

`XLError.index(sub[, start[, end]]) → int`

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum

`XLError.isalnum()`

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha

`XLError.isalpha()`

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii

`XLError.isascii()`

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal

`XLError.isdecimal()`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit

`XLError.isdigit()`

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier

`XLError.isidentifier()`

Return True if the string is a valid Python identifier, False otherwise.

Call `keyword.iskeyword(s)` to test whether string `s` is a reserved identifier, such as “def” or “class”.

islower

XLError.islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric

XLError.isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable

XLError.isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace

XLError.isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle

XLError.istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper

XLError.isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join

XLError.join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

ljust

XLError.ljust(*width*, *fillchar*=' ', /)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower

XLError.lower()

Return a copy of the string converted to lowercase.

lstrip

XLError.lstrip(*chars*=None, /)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

maketrans

static XLError.maketrans()

Return a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition

XLError.partition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

removeprefix

`XLError.removeprefix(prefix, /)`

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return `string[len(prefix):]`. Otherwise, return a copy of the original string.

removesuffix

`XLError.removesuffix(suffix, /)`

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return `string[:-len(suffix)]`. Otherwise, return a copy of the original string.

replace

`XLError.replace(old, new, count=-1, /)`

Return a copy with all occurrences of substring `old` replaced by `new`.

count

Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument `count` is given, only the first `count` occurrences are replaced.

rfind

`XLError.rfind(sub[, start[, end]]) → int`

Return the highest index in `S` where substring `sub` is found, such that `sub` is contained within `S[start:end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

Return -1 on failure.

rindex

`XLError.rindex(sub[, start[, end]]) → int`

Return the highest index in `S` where substring `sub` is found, such that `sub` is contained within `S[start:end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

Raises `ValueError` when the substring is not found.

rjust

`XLError.rjust(width, fillchar=' ', /)`

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition

`XLError.rpartition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit

`XLError.rsplit(sep=None, maxsplit=-1)`

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including `n` `r` `t` `f` and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip

`XLError.rstrip(chars=None, /)`

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split

`XLError.split(sep=None, maxsplit=-1)`

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including `n` `r` `t` `f` and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, `str.split()` is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines

`XLError.splitlines(keepends=False)`

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless `keepends` is given and true.

startswith

`XLError.startswith(prefix[, start[, end]]) → bool`

Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. `prefix` can also be a tuple of strings to try.

strip

`XLError.strip(chars=None, /)`

Return a copy of the string with leading and trailing whitespace removed.

If `chars` is given and not None, remove characters in `chars` instead.

swapcase

`XLError.swapcase()`

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title

`XLError.title()`

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate

`XLError.translate(table, /)`

Replace each character in the string using the given translation table.

table

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper

`XLError.upper()`

Return a copy of the string converted to uppercase.

zfill

`XLError.zfill(width, /)`

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

`__init__(*args)`

operator

It provides Operator classes.

Classes

Intersect

Operator

OperatorToken

Separator

Intersect

`class Intersect(s, context=None)`

Methods

<code>__init__</code>
<code>ast</code>
<code>compile</code>
<code>match</code>
<code>process</code>
<code>set_expr</code>
<code>update_input_tokens</code>
<code>update_name</code>

`__init__`

`Intersect.__init__(s, context=None)`

`ast`

`Intersect.ast(tokens, stack, builder)`

`compile`

`Intersect.compile()`

`match`

`Intersect.match(s)`

`process`

`Intersect.process(match, context=None)`

set_expr

`Intersect.set_expr(*tokens)`

update_input_tokens

`Intersect.update_input_tokens(*tokens)`

update_name

`Intersect.update_name(tokens, stack)`

`__init__(s, context=None)`

Attributes

<code>get_n_args</code>
<code>name</code>
<code>node_id</code>
<code>pred</code>

get_n_args

property `Intersect.get_n_args`

name

property `Intersect.name`

node_id

property `Intersect.node_id`

pred

property `Intersect.pred`

Operator

class `Operator(s, context=None)`

Methods

<code>__init__</code>
<code>ast</code>
<code>compile</code>
<code>match</code>
<code>process</code>
<code>set_expr</code>
<code>update_input_tokens</code>
<code>update_name</code>

`__init__`

`Operator.__init__(s, context=None)`

ast

`Operator.ast(tokens, stack, builder)`

compile

`Operator.compile()`

match

`Operator.match(s)`

process

`Operator.process(match, context=None)`

set_expr

`Operator.set_expr(*tokens)`

update_input_tokens

`Operator.update_input_tokens(*tokens)`

update_name

`Operator.update_name(tokens, stack)`

`__init__(s, context=None)`

Attributes

<code>get_n_args</code>
<code>name</code>
<code>node_id</code>
<code>pred</code>

get_n_args

property `Operator.get_n_args`

name

property `Operator.name`

node_id

property `Operator.node_id`

pred

property `Operator.pred`

OperatorToken

class `OperatorToken(s, context=None)`

Methods

<code>__init__</code>
<code>ast</code>
<code>compile</code>
<code>match</code>
<code>process</code>
<code>set_expr</code>
<code>update_input_tokens</code>
<code>update_name</code>

__init__

`OperatorToken.__init__(s, context=None)`

ast

`OperatorToken.ast(tokens, stack, builder)`

compile

`OperatorToken.compile()`

match

`OperatorToken.match(s)`

process

`OperatorToken.process(match, context=None)`

set_expr

`OperatorToken.set_expr(*tokens)`

update_input_tokens

`OperatorToken.update_input_tokens(*tokens)`

update_name

`OperatorToken.update_name(tokens, stack)`

`__init__(s, context=None)`

Attributes

get_n_args
name
node_id
pred

get_n_args

property OperatorToken.get_n_args

name

property OperatorToken.name

node_id

property OperatorToken.node_id

pred

property OperatorToken.pred

Separator

class Separator(*s*, *context=None*)

Methods

<code>__init__</code>
<code>ast</code>
<code>compile</code>
<code>match</code>
<code>process</code>
<code>set_expr</code>
<code>update_input_tokens</code>
<code>update_name</code>

__init__

Separator.**__init__**(*s*, *context=None*)

ast

Separator.**ast**(*tokens*, *stack*, *builder*)

compile

Separator.**compile**()

match

Separator.**match**(*s*)

process

Separator.**process**(*match*, *context=None*)

set_expr

Separator.**set_expr**(**tokens*)

update_input_tokens

Separator.**update_input_tokens**(**tokens*)

update_name

Separator.**update_name**(*tokens*, *stack*)

__init__(*s*, *context=None*)

Attributes

get_n_args
name
node_id
pred

get_n_args

property Separator.get_n_args

name

property Separator.name

node_id

property Separator.node_id

pred

property Separator.pred

parenthesis

It provides Parenthesis class.

Classes

Parenthesis

Parenthesis

class Parenthesis(*s*, *context=None*)

Methods

__init__

ast

match

process

set_expr

update_input_tokens

`__init__`

`Parenthesis.__init__(s, context=None)`

`ast`

`Parenthesis.ast(tokens, stack, builder)`

`match`

`Parenthesis.match(s)`

`process`

`Parenthesis.process(match, context=None)`

`set_expr`

`Parenthesis.set_expr(*tokens)`

`update_input_tokens`

`Parenthesis.update_input_tokens(*tokens)`

`__init__(s, context=None)`

Attributes

<code>n_args</code>
<code>name</code>
<code>node_id</code>
<code>opens</code>

n_args

`Parenthesis.n_args = 0`

name

property `Parenthesis.name`

node_id

property `Parenthesis.node_id`

opens

`Parenthesis.opens = {')': '('}`

Classes

<i>Token</i>

Token

class `Token(s, context=None)`

Methods

<code>__init__</code>

<code>ast</code>

<code>match</code>

<code>process</code>

<code>set_expr</code>

<code>update_input_tokens</code>

`__init__`

`Token.__init__(s, context=None)`

`ast`

`Token.ast(tokens, stack, builder)`

`match`

`Token.match(s)`

`process`

`Token.process(match, context=None)`

`set_expr`

`Token.set_expr(*tokens)`

`update_input_tokens`

`Token.update_input_tokens(*tokens)`

`__init__(s, context=None)`

Attributes

<code>name</code>
<code>node_id</code>

`name`

property `Token.name`

node_id

property Token.**node_id**

2.2.7.5 functions

It provides functions implementations to compile the Excel functions.

Sub-Modules:

<i>comp</i>	Python equivalents of compatibility Excel functions.
<i>date</i>	Python equivalents of financial Excel functions.
<i>eng</i>	Python equivalents of engineering Excel functions.
<i>financial</i>	Python equivalents of financial Excel functions.
<i>google</i>	Python equivalents of google Excel functions.
<i>info</i>	Python equivalents of information Excel functions.
<i>logic</i>	Python equivalents of logical Excel functions.
<i>look</i>	Python equivalents of lookup and reference Excel functions.
<i>math</i>	Python equivalents of math and trigonometry Excel functions.
<i>operators</i>	Python equivalents of Excel operators.
<i>stat</i>	Python equivalents of statistical Excel functions.
<i>text</i>	Python equivalents of text Excel functions.

comp

Python equivalents of compatibility Excel functions.

date

Python equivalents of financial Excel functions.

Functions

<code>xdate</code>
<code>xdatedif</code>
<code>xdatevalue</code>
<code>xday</code>
<code>xedate</code>
<code>xisoweknum</code>
<code>xnow</code>
<code>xsecond</code>
<code>xtime</code>
<code>xtimevalue</code>
<code>xtoday</code>
<code>xweekday</code>
<code>xweeknum</code>
<code>xyearfrac</code>

xdate

xdate(*year, month, day*)

xdatedif

xdatedif(*start_date, end_date, unit*)

xdatevalue

xdatevalue(*date_text*)

xday

xday(*serial_number*, *n*=2)

xedate

xedate(*start_date*, *months*)

xisoweeknum

xisoweeknum(*serial_number*)

xnow

xnow()

xsecond

xsecond(*serial_number*, *n*=2)

xtime

xtime(*hour*, *minute*, *second*)

xtimevalue

xtimevalue(*time_text*)

xtoday

xtoday()

xweekday

xweekday(*serial_number*, *n*=1)

xweeknum

xweeknum(*serial_number*, *n=1*)

xyearfrac

xyearfrac(*start_date*, *end_date*, *basis=0*)

eng

Python equivalents of engineering Excel functions.

Functions

<code>hex2dec2bin2oct</code>

hex2dec2bin2oct

hex2dec2bin2oct(*function_id*, *memo*)

financial

Python equivalents of financial Excel functions.

Functions

<code>xcumipmt</code>

<code>xirr</code>

<code>xnper</code>

<code>xnpv</code>

<code>xppmt</code>

<code>xrate</code>

<code>xxirr</code>

<code>xxnpv</code>

xcumipmt

xcumipmt(*rate*, *nper*, *pv*, *start_period*, *end_period*, *type*)

xirr

xirr(*values*, *guess*=0.1)

xnper

xnper(*rate*, *pmt*, *pv*, *fv*=0, *type*=0)

xnpv

xnpv(*rate*, *values*, *dates*=None)

xppmt

xppmt(*rate*, *per*, *nper*, *pv*, *fv*=0, *type*=0)

xrate

xrate(*nper*, *pmt*, *pv*, *fv*=0, *type*=0, *guess*=0.1)

xxirr

xxirr(*values*, *dates*, *x*=0.1)

xxnpv

xxnpv(*rate*, *values*, *dates*)

google

Python equivalents of google Excel functions.

Functions

<i>xdummy</i>

xdummy

xdummy(*args)

info

Python equivalents of information Excel functions.

Functions

<i>iserr</i>

<i>iserror</i>

<i>isna</i>

<i>xiseven_odd</i>

<i>xna</i>

iserr

iserr(val)

iserror

iserror(val, check=<function <lambda>>, array=<class 'formulas.functions.info.IsErrorArray'>)

isna

isna(value)

xiseven_odd

xiseven_odd(*number*, *odd=False*)

xna

xna()

Classes

<i>IsErrArray</i>
<i>IsErrorArray</i>
<i>IsNaArray</i>
<i>IsNumberArray</i>

IsErrArray

class IsErrArray

Methods

<i>__init__</i>	
all	Returns True if all elements evaluate to True.
any	Returns True if any of the elements of <i>a</i> evaluate to True.
argmax	Return indices of the maximum values along the given axis.
argmin	Return indices of the minimum values along the given axis.
argpartition	Returns the indices that would partition this array.
argsort	Returns the indices that would sort this array.
astype	Copy of the array, cast to a specified type.
byteswap	Swap the bytes of the array elements
choose	Use an index array to construct a new array from a set of choices.
clip	Return an array whose values are limited to [min, max].
collapse	
compress	Return selected slices of this array along given axis.
conj	Complex-conjugate all elements.

continues on next page

Table 2 – continued from previous page

conjugate	Return the complex conjugate, element-wise.
copy	Return a copy of the array.
cumprod	Return the cumulative product of the elements along the given axis.
cumsum	Return the cumulative sum of the elements along the given axis.
diagonal	Return specified diagonals.
dot	
dump	Dump a pickle of the array to the specified file.
dumps	Returns the pickle of the array as a string.
fill	Fill the array with a scalar value.
flatten	Return a copy of the array collapsed into one dimension.
getfield	Returns a field of the given array as a certain type.
item	Copy an element of an array to a standard Python scalar and return it.
itemset	Insert scalar into an array (scalar is cast to array's dtype, if possible)
max	Return the maximum along a given axis.
mean	Returns the average of the array elements along given axis.
min	Return the minimum along a given axis.
newbyteorder	Return the array with the same data viewed with a different byte order.
nonzero	Return the indices of the elements that are non-zero.
partition	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
prod	Return the product of the array elements over the given axis
ptp	Peak to peak (maximum - minimum) value along a given axis.
put	Set <code>a.flat[n] = values[n]</code> for all <code>n</code> in indices.
ravel	Return a flattened array.
repeat	Repeat elements of an array.
reshape	Returns an array containing the same data with a new shape.
resize	Change shape and size of array in-place.
round	Return <code>a</code> with each element rounded to the given number of decimals.
searchsorted	Find indices where elements of <code>v</code> should be inserted in <code>a</code> to maintain order.
setfield	Put a value into a specified place in a field defined by a data-type.
setflags	Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY, respectively.
sort	Sort an array in-place.
squeeze	Remove axes of length one from <code>a</code> .
std	Returns the standard deviation of the array elements along given axis.

continues on next page

Table 2 – continued from previous page

<code>sum</code>	Return the sum of the array elements over the given axis.
<code>swapaxes</code>	Return a view of the array with <i>axis1</i> and <i>axis2</i> interchanged.
<code>take</code>	Return an array formed from the elements of <i>a</i> at the given indices.
<code>tobytes</code>	Construct Python bytes containing the raw data bytes in the array.
<code>tofile</code>	Write array to a file as text or binary (default).
<code>tolist</code>	Return the array as an <code>a.ndim</code> -levels deep nested list of Python scalars.
<code>tostring</code>	A compatibility alias for <i>tobytes</i> , with exactly the same behavior.
<code>trace</code>	Return the sum along diagonals of the array.
<code>transpose</code>	Returns a view of the array with axes transposed.
<code>var</code>	Returns the variance of the array elements, along given axis.
<code>view</code>	New view of array with the same data.

`__init__`

`IsErrArray.__init__()`

`all`

`IsErrArray.all(axis=None, out=None, keepdims=False, *, where=True)`

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

See Also

`numpy.all` : equivalent function

`any`

`IsErrArray.any(axis=None, out=None, keepdims=False, *, where=True)`

Returns True if any of the elements of *a* evaluate to True.

Refer to *numpy.any* for full documentation.

See Also

`numpy.any` : equivalent function

`argmax`

`IsErrArray.argmax(axis=None, out=None, *, keepdims=False)`

Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

See Also

`numpy.argmax` : equivalent function

`argmin`

`IsErrArray.argmin(axis=None, out=None, *, keepdims=False)`

Return indices of the minimum values along the given axis.

Refer to `numpy.argmin` for detailed documentation.

See Also

`numpy.argmin` : equivalent function

`argpartition`

`IsErrArray.argpartition(kth, axis=-1, kind='introselect', order=None)`

Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

See Also

`numpy.argpartition` : equivalent function

argsort

`IsErrArray.argsort(axis=-1, kind=None, order=None)`

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See Also

`numpy.argsort` : equivalent function

astype

`IsErrArray.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)`

Copy of the array, cast to a specified type.

Parameters

dtype

[str or dtype] Typecode or data-type to which the array is cast.

order

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok

[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy

[bool, optional] By default, `astype` always returns a newly allocated array. If this is set to false, and the `dtype`, `order`, and `subok` requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t

[ndarray] Unless *copy* is False and the other conditions for returning the input array are satisfied (see description for *copy* input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by *dtype*, *order*.

Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for “unsafe” casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in ‘safe’ casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

Raises

ComplexWarning

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

byteswap

`IsErrArray.byteswap(inplace=False)`

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

Parameters

inplace

[bool, optional] If True, swap bytes in-place, default is False.

Returns

out

[ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([ 256,      1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='<S3')
```

`A.newbyteorder().byteswap()` produces an array with the same values
but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

choose

`IsErrArray.choose(choices, out=None, mode='raise')`

Use an index array to construct a new array from a set of choices.

Refer to *numpy.choose* for full documentation.

See Also

`numpy.choose` : equivalent function

clip

`IsErrArray.clip(min=None, max=None, out=None, **kwargs)`

Return an array whose values are limited to `[min, max]`. One of `max` or `min` must be given.

Refer to `numpy.clip` for full documentation.

See Also

`numpy.clip` : equivalent function

collapse

`IsErrArray.collapse(shape)`

compress

`IsErrArray.compress(condition, axis=None, out=None)`

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also

`numpy.compress` : equivalent function

conj

`IsErrArray.conj()`

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See Also

`numpy.conjugate` : equivalent function

conjugate

`IsErrArray.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

See Also

`numpy.conjugate` : equivalent function

copy

`IsErrArray.copy(order='C')`

Return a copy of the array.

Parameters

order

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy()` are very similar but have different default values for their `order=` arguments, and this function always passes sub-classes through.)

See also

`numpy.copy` : Similar function with different default behavior `numpy.copyto`

Notes

This function is the preferred method for creating an array copy. The function `numpy.copy()` is similar, but it defaults to using order 'K', and will not pass sub-classes through by default.

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y  
array([[1, 2, 3],  
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']  
True
```

cumprod

`IsErrArray.cumprod(axis=None, dtype=None, out=None)`

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See Also

`numpy.cumprod` : equivalent function

cumsum

`IsErrArray.cumsum(axis=None, dtype=None, out=None)`

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See Also

`numpy.cumsum` : equivalent function

diagonal

`IsErrArray.diagonal(offset=0, axis1=0, axis2=1)`

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal()` for full documentation.

See Also

`numpy.diagonal` : equivalent function

dot

`IsErrArray.dot()`

dump

`IsErrArray.dump(file)`

Dump a pickle of the array to the specified file. The array can be read back with `pickle.load` or `numpy.load`.

Parameters**file**

[str or Path] A string naming the dump file.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

dumps

`IsErrArray.dumps()`

Returns the pickle of the array as a string. `pickle.loads` will convert the string back to an array.

Parameters

None

fill

`IsErrArray.fill(value)`

Fill the array with a scalar value.

Parameters**value**

[scalar] All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.])
```


Fill expects a scalar value and always behaves the same as assigning to a single array element. The following is a rare example where this distinction is important:

```
>>> a = np.array([None, None], dtype=object)
>>> a[0] = np.array(3)
>>> a
array([array(3), None], dtype=object)
>>> a.fill(np.array(3))
>>> a
array([array(3), array(3)], dtype=object)
```

Where other forms of assignments will unpack the array being assigned:

```
>>> a[...] = np.array(3)
>>> a
array([3, 3], dtype=object)
```

flatten

`IsErrArray.flatten(order='C')`

Return a copy of the array collapsed into one dimension.

Parameters

order

[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns

y

[ndarray] A copy of the input array, flattened to one dimension.

See Also

`ravel` : Return a flattened array. `flat` : A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

getfield

`IsErrArray.getfield(dtype, offset=0)`

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype `complex128` has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype

[str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset

[int] Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

item

`IsErrArray.item(*args)`

Copy an element of an array to a standard Python scalar and return it.

Parameters

*args : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z

[Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to *a[args]*, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

itemset

`IsErrArray.itemset(*args)`

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

***args**

[Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

max

`IsErrArray.max(axis=None, out=None, keepdims=False, initial=<no value>, where=True)`

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

See Also

`numpy.amax` : equivalent function

mean

`IsErrArray.mean(axis=None, dtype=None, out=None, keepdims=False, *, where=True)`

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See Also

`numpy.mean` : equivalent function

min

`IsErrArray.min(axis=None, out=None, keepdims=False, initial=<no value>, where=True)`

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See Also

`numpy.amin` : equivalent function

newbyteorder

`IsErrArray.newbyteorder(new_order='S', /)`

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters

`new_order`

[string, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- ‘S’ - swap dtype from current to opposite endian
- {‘<’, ‘little’} - little endian
- {‘>’, ‘big’} - big endian
- {‘=’, ‘native’} - native order, equivalent to `sys.byteorder`

- {'I', 'T'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order.

Returns

new_arr

[array] New array object with the dtype reflecting given change to the byte order.

nonzero

`IsErrArray.nonzero()`

Return the indices of the elements that are non-zero.

Refer to *numpy.nonzero* for full documentation.

See Also

`numpy.nonzero` : equivalent function

partition

`IsErrArray.partition(kth, axis=-1, kind='introselect', order=None)`

Rearranges the elements in the array in such a way that the value of the element in *kth* position is in the position it would be in a sorted array. All elements smaller than the *kth* element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

Parameters

kth

[int or sequence of ints] Element index to partition by. The *kth* element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of *kth* it will partition all elements indexed by *kth* of them into their sorted position at once.

Deprecated since version 1.22.0: Passing booleans as index is deprecated.

axis

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind

[{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

order

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

`numpy.partition` : Return a partitioned copy of an array. `argsort` : Indirect partition. `sort` : Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

prod

`IsErrArray.prod(axis=None, dtype=None, out=None, keepdims=False, initial=1, where=True)`

Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See Also

`numpy.prod` : equivalent function

ptp

`IsErrArray.ptp(axis=None, out=None, keepdims=False)`

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See Also

`numpy.ptp` : equivalent function

put

`IsErrArray.put(indices, values, mode='raise')`

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *numpy.put* for full documentation.

See Also

`numpy.put` : equivalent function

ravel

`IsErrArray.ravel([order])`

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

See Also

`numpy.ravel` : equivalent function

`ndarray.flat` : a flat iterator on the array.

repeat

`IsErrArray.repeat(repeats, axis=None)`

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

See Also

`numpy.repeat` : equivalent function

reshape

`IsErrArray.reshape(shape, order='C')`

Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

See Also

`numpy.reshape` : equivalent function

Notes

Unlike the free function *numpy.reshape*, this method on *ndarray* allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

resize

`IsErrArray.resize(new_shape, refcheck=True)`

Change shape and size of array in-place.

Parameters

new_shape

[tuple of ints, or *n* ints] Shape of resized array.

refcheck

[bool, optional] If False, reference count will not be checked. Default is True.

Returns

None

Raises

ValueError

If *a* does not own its own data or references or views to it exist, and the data memory must be changed.
PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

SystemError

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See Also

`resize` : Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and re-shaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

round

`IsErrArray.round(decimals=0, out=None)`

Return *a* with each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

See Also

`numpy.around` : equivalent function

searchsorted

`IsErrArray.searchsorted(v, side='left', sorter=None)`

Find indices where elements of *v* should be inserted in *a* to maintain order.

For full documentation, see `numpy.searchsorted`

See Also

`numpy.searchsorted` : equivalent function

setfield

`IsErrArray.setfield(val, dtype, offset=0)`

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters

val

[object] Value to be placed in field.

dtype

[dtype object] Data-type of the field in which to place *val*.

offset

[int, optional] The number of bytes into the field at which to place *val*.

Returns

None

See Also

`getfield`

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

setflags

`IsErrArray.setflags(write=None, align=None, uic=None)`

Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write

[bool, optional] Describes whether or not *a* can be written to.

align

[bool, optional] Describes whether or not *a* is aligned properly for its type.

uic

[bool, optional] Describes whether or not *a* is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: `WRITEBACKIFCOPY`, `WRITEABLE`, and `ALIGNED`.

`WRITEABLE` (W) the data area can be written to;

`ALIGNED` (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

`WRITEBACKIFCOPY` (X) this array is a copy of some other array (referenced by `.base`). When the C-API function `PyArray_ResolveWritebackIfCopy` is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
WRITEBACKIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

sort

`IsErrArray.sort(axis=-1, kind=None, order=None)`

Sort an array in-place. Refer to *numpy.sort* for full documentation.

Parameters

axis

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0: The 'stable' option was added.

order

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

`numpy.sort` : Return a sorted copy of an array. `numpy.argsort` : Indirect sort. `numpy.lexsort` : Indirect stable sort on multiple keys. `numpy.searchsorted` : Find elements in sorted array. `numpy.partition`: Partial sort.

Notes

See *numpy.sort* for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

squeeze

`IsErrArray.squeeze(axis=None)`

Remove axes of length one from *a*.

Refer to `numpy.squeeze` for full documentation.

See Also

`numpy.squeeze` : equivalent function

std

`IsErrArray.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True)`

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See Also

`numpy.std` : equivalent function

sum

`IsErrArray.sum(axis=None, dtype=None, out=None, keepdims=False, initial=0, where=True)`

Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See Also

`numpy.sum` : equivalent function

swapaxes

`IsErrArray.swapaxes(axis1, axis2)`

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *numpy.swapaxes* for full documentation.

See Also

`numpy.swapaxes` : equivalent function

take

`IsErrArray.take(indices, axis=None, out=None, mode='raise')`

Return an array formed from the elements of *a* at the given indices.

Refer to *numpy.take* for full documentation.

See Also

`numpy.take` : equivalent function

tobytes

`IsErrArray.tobytes(order='C')`

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object is produced in C-order by default. This behavior is controlled by the `order` parameter.

New in version 1.9.0.

Parameters

order

[['C', 'F', 'A'], optional] Controls the memory layout of the bytes object. 'C' means C-order, 'F' means F-order, 'A' (short for *Any*) means 'F' if *a* is Fortran contiguous, 'C' otherwise. Default is 'C'.

Returns

s
[bytes] Python bytes exhibiting a copy of *a*'s raw data.

See also

frombuffer

Inverse of this operation, construct a 1-dimensional array from Python bytes.

Examples

```

>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'

```

tofile

`IsErrArray.tofile(fid, sep="", format='%s')`

Write array to a file as text or binary (default).

Data is always written in ‘C’ order, independent of the order of *a*. The data produced by this method can be recovered using the function `fromfile()`.

Parameters

fid

[file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

sep

[str] Separator between array items for text output. If “” (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format

[str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When *fid* is a file object, array contents are directly written to the file, bypassing the file object’s `write` method. As a result, `tofile` cannot be used with files objects supporting compression (e.g., `GzipFile`) or file-like objects that do not support `fileno()` (e.g., `BytesIO`).

tolist

`IsErrArray.tolist()`

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the `~numpy.ndarray.item` function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

Parameters

none

Returns

`y`
[object, or list of object, or list of list of object, or ...] The possibly nested list of array elements.

Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

tostring

`IsErrArray.tostring(order='C')`

A compatibility alias for *tobytes*, with exactly the same behavior.

Despite its name, it returns *bytes* not *strs*.

Deprecated since version 1.19.0.

trace

`IsErrArray.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Return the sum along diagonals of the array.

Refer to *numpy.trace* for full documentation.

See Also

`numpy.trace` : equivalent function

transpose

`IsErrArray.transpose(*axes)`

Returns a view of the array with axes transposed.

Refer to *numpy.transpose* for full documentation.

Parameters

axes : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means that the array's *i*-th axis becomes the transposed array's *j*-th axis.
- *n* ints: same as an *n*-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form).

Returns

p
[ndarray] View of the array with its axes suitably permuted.

See Also

`transpose` : Equivalent function. `ndarray.T` : Array property returning the array transposed. `ndarray.reshape` : Give a new shape to an array without changing its data.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
>>> a.transpose()
array([1, 2, 3, 4])
```

var

`IsErrArray.var`(*axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True*)

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See Also

`numpy.var` : equivalent function

view

`IsErrArray.view([dtype][, type])`

New view of array with the same data.

Note: Passing `None` for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

Parameters

dtype

[data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., `float32` or `int16`. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

type

[Python type, optional] Type of the returned view, e.g., `ndarray` or `matrix`. Again, omission of the parameter results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the last axis of *a* must be contiguous. This axis will be resized in the result.

Changed in version 1.23.0: Only the last axis needs to be contiguous. Previously, the entire array had to be C-contiguous.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2), (3, 4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1, 2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2., 3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0, 1] = 20
>>> x
array([(1, 20), (3, 4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int16)
>>> y = x[:, ::2]
>>> y
array([[1, 3],
       [4, 6]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
...
ValueError: To change to a dtype of a different size, the last axis must be
↳ contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[1, 3],
       [4, 6]], dtype=[('width', '<i2'), ('length', '<i2')])
```

However, views that change dtype are totally fine for arrays with a contiguous last axis, even if the rest of the axes are not C-contiguous:

```
>>> x = np.arange(2 * 3 * 4, dtype=np.int8).reshape(2, 3, 4)
>>> x.transpose(1, 0, 2).view(np.int16)
array([[256, 770],
       [3340, 3854]],
      [[1284, 1798],
       [4368, 4882]],
```

(continues on next page)

(continued from previous page)

```
[[2312, 2826],
 [5396, 5910]], dtype=int16)
```

`__init__()`

Attributes

<code>T</code>	View of the transposed array.
<code>base</code>	Base object if memory is from some other object.
<code>ctypes</code>	An object to simplify the interaction of the array with the ctypes module.
<code>data</code>	Python buffer object pointing to the start of the array's data.
<code>dtype</code>	Data-type of the array's elements.
<code>flags</code>	Information about the memory layout of the array.
<code>flat</code>	A 1-D iterator over the array.
<code>imag</code>	The imaginary part of the array.
<code>itemsize</code>	Length of one array element in bytes.
<code>nbytes</code>	Total bytes consumed by the elements of the array.
<code>ndim</code>	Number of array dimensions.
<code>real</code>	The real part of the array.
<code>shape</code>	Tuple of array dimensions.
<code>size</code>	Number of elements in the array.
<code>strides</code>	Tuple of bytes to step in each dimension when traversing an array.

T

`IsErrArray.T`

View of the transposed array.

Same as `self.transpose()`.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.T
array([[1, 3],
       [2, 4]])
```

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
```

(continues on next page)

(continued from previous page)

```
>>> a.T  
array([1, 2, 3, 4])
```

See Also

`transpose`

base

`IsErrArray.base`

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is `None`:

```
>>> x = np.array([1,2,3,4])  
>>> x.base is None  
True
```

Slicing creates a view, whose memory is shared with `x`:

```
>>> y = x[2:]  
>>> y.base is x  
True
```

ctypes

`IsErrArray.ctypes`

An object to simplify the interaction of the array with the `ctypes` module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the `ctypes` module. The returned object has, among others, `data`, `shape`, and `strides` attributes (see Notes below) which themselves return `ctypes` objects that can be used as arguments to a shared library.

Parameters

`None`

Returns

c

[Python object] Possessing attributes data, shape, strides, etc.

See Also

`numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

`_ctypes.data`

A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

`_ctypes.shape`

(`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the C-integer corresponding to `dtype('p')` on this platform (see `~numpy.ctypeslib.c_intp`). This base-type could be `ctypes.c_int`, `ctypes.c_long`, or `ctypes.c_longlong` depending on the platform. The ctypes array contains the shape of the underlying array.

`_ctypes.strides`

(`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

`_ctypes.data_as(obj)`

Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

`_ctypes.shape_as(obj)`

Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

`_ctypes.strides_as(obj)`

Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

data

`IsErrArray.data`

Python buffer object pointing to the start of the array's data.

dtype

`IsErrArray.dtype`

Data-type of the array's elements.

Warning: Setting `arr.dtype` is discouraged and may be deprecated in the future. Setting will replace the dtype without modifying the memory (see also `ndarray.view` and `ndarray.astype`).

Parameters

None

Returns

d : numpy dtype object

See Also

ndarray.astype : Cast the values contained in the array to a new data-type. ndarray.view : Create a view of the same data but a different data-type. numpy.dtype

Examples

```

>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>

```

flags

IsErrArray.flags

Information about the memory layout of the array.

Attributes

C_CONTIGUOUS (C)

The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F)

The data is in a single, Fortran-style contiguous segment.

OWNDATA (O)

The array owns the memory it uses or borrows it from another object.

WRITEABLE (W)

The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

ALIGNED (A)

The data and all elements are aligned appropriately for the hardware.

WRITEBACKIFCOPY (X)

This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

FNC

F_CONTIGUOUS and not C_CONTIGUOUS.

FORC

F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B)

ALIGNED and WRITEABLE.

CARRAY (CA)

BEHAVED and C_CONTIGUOUS.

FARRAY (FA)

BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the `WRITEBACKIFCOPY`, `WRITEABLE`, and `ALIGNED` flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- `WRITEBACKIFCOPY` can only be set `False`.
- `ALIGNED` can only be set `True` if the data is truly aligned.
- `WRITEABLE` can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

flat

IsErrArray.flat

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See Also

`flatten` : Return a copy of the array collapsed into one dimension.

`flatiter`

Examples

```

>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>

```

An assignment example:

```

>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])

```

imag

`IsErrArray.imag`

The imaginary part of the array.

Examples

```

>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.          ,  0.70710678])
>>> x.imag.dtype
dtype('float64')

```

itemsize

`IsErrArray.itemsize`

Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

nbytes

`IsErrArray.nbytes`

Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

See Also

`sys.getsizeof`

Memory consumed by the object itself without parents in case view. This does include memory consumed by non-element attributes.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim

IsErrArray.ndim

Number of array dimensions.

Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

real

IsErrArray.real

The real part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.          ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

See Also

`numpy.real` : equivalent function

shape

IsErrArray.shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with *numpy.reshape*, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

Warning: Setting `arr.shape` is discouraged and may be deprecated in the future. Using *ndarray.reshape* is the preferred approach.

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[:,2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

See Also

`numpy.shape` : Equivalent getter function. `numpy.reshape` : Function similar to setting `shape`. `ndarray.reshape` : Method similar to setting `shape`.

size

`IsErrArray.size`

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Notes

`a.size` returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

strides

IsErrArray.strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element ($i[0]$, $i[1]$, ..., $i[n]$) in an array a is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

Warning: Setting `arr.strides` is discouraged and may be deprecated in the future. `numpy.lib.stride_tricks.as_strided` should be preferred to create a new view of the same data in a safer way.

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array x will be (20, 4).

See Also

`numpy.lib.stride_tricks.as_strided`

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

IsErrorArray

class IsErrorArray

Methods

<code>__init__</code>	
<code>all</code>	Returns True if all elements evaluate to True.
<code>any</code>	Returns True if any of the elements of <i>a</i> evaluate to True.
<code>argmax</code>	Return indices of the maximum values along the given axis.
<code>argmin</code>	Return indices of the minimum values along the given axis.
<code>argpartition</code>	Returns the indices that would partition this array.
<code>argsort</code>	Returns the indices that would sort this array.
<code>astype</code>	Copy of the array, cast to a specified type.
<code>byteswap</code>	Swap the bytes of the array elements
<code>choose</code>	Use an index array to construct a new array from a set of choices.

continues on next page

Table 3 – continued from previous page

clip	Return an array whose values are limited to [min, max].
collapse	
compress	Return selected slices of this array along given axis.
conj	Complex-conjugate all elements.
conjugate	Return the complex conjugate, element-wise.
copy	Return a copy of the array.
cumprod	Return the cumulative product of the elements along the given axis.
cumsum	Return the cumulative sum of the elements along the given axis.
diagonal	Return specified diagonals.
dot	
dump	Dump a pickle of the array to the specified file.
dumps	Returns the pickle of the array as a string.
fill	Fill the array with a scalar value.
flatten	Return a copy of the array collapsed into one dimension.
getfield	Returns a field of the given array as a certain type.
item	Copy an element of an array to a standard Python scalar and return it.
itemset	Insert scalar into an array (scalar is cast to array's dtype, if possible)
max	Return the maximum along a given axis.
mean	Returns the average of the array elements along given axis.
min	Return the minimum along a given axis.
newbyteorder	Return the array with the same data viewed with a different byte order.
nonzero	Return the indices of the elements that are non-zero.
partition	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
prod	Return the product of the array elements over the given axis
ptp	Peak to peak (maximum - minimum) value along a given axis.
put	Set <code>a.flat[n] = values[n]</code> for all <code>n</code> in indices.
ravel	Return a flattened array.
repeat	Repeat elements of an array.
reshape	Returns an array containing the same data with a new shape.
resize	Change shape and size of array in-place.
round	Return <code>a</code> with each element rounded to the given number of decimals.
searchsorted	Find indices where elements of <code>v</code> should be inserted in <code>a</code> to maintain order.
setfield	Put a value into a specified place in a field defined by a data-type.

continues on next page

Table 3 – continued from previous page

<code>setflags</code>	Set array flags <code>WRITEABLE</code> , <code>ALIGNED</code> , <code>WRITEBACKIFCOPY</code> , respectively.
<code>sort</code>	Sort an array in-place.
<code>squeeze</code>	Remove axes of length one from <i>a</i> .
<code>std</code>	Returns the standard deviation of the array elements along given axis.
<code>sum</code>	Return the sum of the array elements over the given axis.
<code>swapaxes</code>	Return a view of the array with <i>axis1</i> and <i>axis2</i> interchanged.
<code>take</code>	Return an array formed from the elements of <i>a</i> at the given indices.
<code>tobytes</code>	Construct Python bytes containing the raw data bytes in the array.
<code>tofile</code>	Write array to a file as text or binary (default).
<code>tolist</code>	Return the array as an <code>a.ndim</code> -levels deep nested list of Python scalars.
<code>tostring</code>	A compatibility alias for <i>tobytes</i> , with exactly the same behavior.
<code>trace</code>	Return the sum along diagonals of the array.
<code>transpose</code>	Returns a view of the array with axes transposed.
<code>var</code>	Returns the variance of the array elements, along given axis.
<code>view</code>	New view of array with the same data.

`__init__`

`IsErrorArray.__init__()`

`all`

`IsErrorArray.all(axis=None, out=None, keepdims=False, *, where=True)`

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

See Also

`numpy.all` : equivalent function

any

`IsErrorArray.any(axis=None, out=None, keepdims=False, *, where=True)`

Returns True if any of the elements of *a* evaluate to True.

Refer to `numpy.any` for full documentation.

See Also

`numpy.any` : equivalent function

argmax

`IsErrorArray.argmax(axis=None, out=None, *, keepdims=False)`

Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

See Also

`numpy.argmax` : equivalent function

argmin

`IsErrorArray.argmin(axis=None, out=None, *, keepdims=False)`

Return indices of the minimum values along the given axis.

Refer to `numpy.argmin` for detailed documentation.

See Also

`numpy.argmin` : equivalent function

argpartition

`IsErrorArray.argpartition(kth, axis=-1, kind='introselect', order=None)`

Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

See Also

`numpy.argmax` : equivalent function

argsort

`IsErrorArray.argsort(axis=-1, kind=None, order=None)`

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See Also

`numpy.argsort` : equivalent function

astype

`IsErrorArray.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)`

Copy of the array, cast to a specified type.

Parameters

dtype

[str or dtype] Typecode or data-type to which the array is cast.

order

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok

[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy

[bool, optional] By default, `astype` always returns a newly allocated array. If this is set to false, and the `dtype`, `order`, and `subok` requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t

[ndarray] Unless *copy* is False and the other conditions for returning the input array are satisfied (see description for *copy* input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by *dtype*, *order*.

Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for “unsafe” casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in ‘safe’ casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

Raises

ComplexWarning

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

byteswap

`IsErrorArray.byteswap(inplace=False)`

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

Parameters

inplace

[bool, optional] If True, swap bytes in-place, default is False.

Returns

out

[ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([ 256,      1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='<S3')
```

`A.newbyteorder().byteswap()` produces an array with the same values
but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

choose

`IsErrorArray.choose(choices, out=None, mode='raise')`

Use an index array to construct a new array from a set of choices.

Refer to *numpy.choose* for full documentation.

See Also

`numpy.choose` : equivalent function

clip

`IsErrorArray.clip(min=None, max=None, out=None, **kwargs)`

Return an array whose values are limited to `[min, max]`. One of `max` or `min` must be given.

Refer to `numpy.clip` for full documentation.

See Also

`numpy.clip` : equivalent function

collapse

`IsErrorArray.collapse(shape)`

compress

`IsErrorArray.compress(condition, axis=None, out=None)`

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also

`numpy.compress` : equivalent function

conj

`IsErrorArray.conj()`

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See Also

`numpy.conjugate` : equivalent function

conjugate

`IsErrorArray.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

See Also

`numpy.conjugate` : equivalent function

copy

`IsErrorArray.copy(order='C')`

Return a copy of the array.

Parameters

order

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy()` are very similar but have different default values for their `order=` arguments, and this function always passes sub-classes through.)

See also

`numpy.copy` : Similar function with different default behavior `numpy.copyto`

Notes

This function is the preferred method for creating an array copy. The function `numpy.copy()` is similar, but it defaults to using order 'K', and will not pass sub-classes through by default.

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y  
array([[1, 2, 3],  
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']  
True
```

cumprod

`IsErrorArray.cumprod(axis=None, dtype=None, out=None)`

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See Also

`numpy.cumprod` : equivalent function

cumsum

`IsErrorArray.cumsum(axis=None, dtype=None, out=None)`

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See Also

`numpy.cumsum` : equivalent function

diagonal

`IsErrorArray.diagonal(offset=0, axis1=0, axis2=1)`

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal()` for full documentation.

See Also

`numpy.diagonal` : equivalent function

dot

`IsErrorArray.dot()`

dump

`IsErrorArray.dump(file)`

Dump a pickle of the array to the specified file. The array can be read back with `pickle.load` or `numpy.load`.

Parameters**file**

[str or Path] A string naming the dump file.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

dumps

`IsErrorArray.dumps()`

Returns the pickle of the array as a string. `pickle.loads` will convert the string back to an array.

Parameters

None

fill

`IsErrorArray.fill(value)`

Fill the array with a scalar value.

Parameters**value**

[scalar] All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.])
```

Fill expects a scalar value and always behaves the same as assigning to a single array element. The following is a rare example where this distinction is important:

```
>>> a = np.array([None, None], dtype=object)
>>> a[0] = np.array(3)
>>> a
array([array(3), None], dtype=object)
>>> a.fill(np.array(3))
>>> a
array([array(3), array(3)], dtype=object)
```

Where other forms of assignments will unpack the array being assigned:

```
>>> a[...] = np.array(3)
>>> a
array([3, 3], dtype=object)
```

flatten

`IsErrorArray.flatten(order='C')`

Return a copy of the array collapsed into one dimension.

Parameters

order

[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns

y

[ndarray] A copy of the input array, flattened to one dimension.

See Also

`ravel` : Return a flattened array. `flat` : A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

getfield

`IsErrorArray.getfield(dtype, offset=0)`

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype `complex128` has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype

[str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset

[int] Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

item

`IsErrorArray.item(*args)`

Copy an element of an array to a standard Python scalar and return it.

Parameters

*args : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z

[Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to *a[args]*, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

itemset

`IsErrorArray.itemset(*args)`

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

***args**

[Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

max

`IsErrorArray.max(axis=None, out=None, keepdims=False, initial=<no value>, where=True)`

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

See Also

`numpy.amax` : equivalent function

mean

`IsErrorArray.mean(axis=None, dtype=None, out=None, keepdims=False, *, where=True)`

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See Also

`numpy.mean` : equivalent function

min

`IsErrorArray.min(axis=None, out=None, keepdims=False, initial=<no value>, where=True)`

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See Also

`numpy.amin` : equivalent function

newbyteorder

`IsErrorArray.newbyteorder(new_order='S', /)`

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters

`new_order`

[string, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- ‘S’ - swap dtype from current to opposite endian
- {‘<’, ‘little’} - little endian
- {‘>’, ‘big’} - big endian
- {‘=’, ‘native’} - native order, equivalent to `sys.byteorder`

- {'I', 'T'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order.

Returns

new_arr

[array] New array object with the dtype reflecting given change to the byte order.

nonzero

`IsErrorArray.nonzero()`

Return the indices of the elements that are non-zero.

Refer to *numpy.nonzero* for full documentation.

See Also

`numpy.nonzero` : equivalent function

partition

`IsErrorArray.partition(kth, axis=-1, kind='introselect', order=None)`

Rearranges the elements in the array in such a way that the value of the element in *kth* position is in the position it would be in a sorted array. All elements smaller than the *kth* element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

Parameters

kth

[int or sequence of ints] Element index to partition by. The *kth* element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of *kth* it will partition all elements indexed by *kth* of them into their sorted position at once.

Deprecated since version 1.22.0: Passing booleans as index is deprecated.

axis

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind

[{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

order

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

`numpy.partition` : Return a partitioned copy of an array. `argsort` : Indirect partition. `sort` : Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

prod

`IsErrorArray.prod(axis=None, dtype=None, out=None, keepdims=False, initial=1, where=True)`

Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See Also

`numpy.prod` : equivalent function

ptp

`IsErrorArray.ptp(axis=None, out=None, keepdims=False)`

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See Also

`numpy.ptp` : equivalent function

put

`IsErrorArray.put(indices, values, mode='raise')`

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *numpy.put* for full documentation.

See Also

`numpy.put` : equivalent function

ravel

`IsErrorArray.ravel([order])`

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

See Also

`numpy.ravel` : equivalent function

`ndarray.flat` : a flat iterator on the array.

repeat

`IsErrorArray.repeat(repeats, axis=None)`

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

See Also

`numpy.repeat` : equivalent function

reshape

`IsErrorArray.reshape(shape, order='C')`

Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

See Also

`numpy.reshape` : equivalent function

Notes

Unlike the free function *numpy.reshape*, this method on *ndarray* allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

resize

`IsErrorArray.resize(new_shape, refcheck=True)`

Change shape and size of array in-place.

Parameters

new_shape

[tuple of ints, or *n* ints] Shape of resized array.

refcheck

[bool, optional] If False, reference count will not be checked. Default is True.

Returns

None

Raises

ValueError

If *a* does not own its own data or references or views to it exist, and the data memory must be changed.
PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

SystemError

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See Also

`resize` : Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and re-shaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

round

`IsErrorArray.round(decimals=0, out=None)`

Return *a* with each element rounded to the given number of decimals.

Refer to *numpy.around* for full documentation.

See Also

`numpy.around` : equivalent function

searchsorted

`IsErrorArray.searchsorted(v, side='left', sorter=None)`

Find indices where elements of *v* should be inserted in *a* to maintain order.

For full documentation, see *numpy.searchsorted*

See Also

`numpy.searchsorted` : equivalent function

setfield

`IsErrorArray.setfield(val, dtype, offset=0)`

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters

val

[object] Value to be placed in field.

dtype

[dtype object] Data-type of the field in which to place *val*.

offset

[int, optional] The number of bytes into the field at which to place *val*.

Returns

None

See Also

`getfield`

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

setflags

`IsErrorArray.setflags(write=None, align=None, uic=None)`

Set array flags `WRITEABLE`, `ALIGNED`, `WRITEBACKIFCOPY`, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The `ALIGNED` flag can only be set to `True` if the data is actually aligned according to the type. The `WRITEBACKIFCOPY` and flag can never be set to `True`. The flag `WRITEABLE` can only be set to `True` if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write

[bool, optional] Describes whether or not *a* can be written to.

align

[bool, optional] Describes whether or not *a* is aligned properly for its type.

uic

[bool, optional] Describes whether or not *a* is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: `WRITEBACKIFCOPY`, `WRITEABLE`, and `ALIGNED`.

`WRITEABLE` (W) the data area can be written to;

`ALIGNED` (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

`WRITEBACKIFCOPY` (X) this array is a copy of some other array (referenced by `.base`). When the C-API function `PyArray_ResolveWritebackIfCopy` is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
WRITEBACKIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

sort

`IsErrorArray.sort(axis=-1, kind=None, order=None)`

Sort an array in-place. Refer to *numpy.sort* for full documentation.

Parameters

axis

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0: The 'stable' option was added.

order

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

`numpy.sort` : Return a sorted copy of an array. `numpy.argsort` : Indirect sort. `numpy.lexsort` : Indirect stable sort on multiple keys. `numpy.searchsorted` : Find elements in sorted array. `numpy.partition`: Partial sort.

Notes

See *numpy.sort* for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([(b'a', 2), (b'c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

squeeze

`IsErrorArray.squeeze(axis=None)`

Remove axes of length one from *a*.

Refer to `numpy.squeeze` for full documentation.

See Also

`numpy.squeeze` : equivalent function

std

`IsErrorArray.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True)`

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See Also

`numpy.std` : equivalent function

sum

`IsErrorArray.sum(axis=None, dtype=None, out=None, keepdims=False, initial=0, where=True)`

Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See Also

`numpy.sum` : equivalent function

swapaxes

`IsErrorArray.swapaxes(axis1, axis2)`

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also

`numpy.swapaxes` : equivalent function

take

`IsErrorArray.take(indices, axis=None, out=None, mode='raise')`

Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

See Also

`numpy.take` : equivalent function

tobytes

`IsErrorArray.tobytes(order='C')`

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object is produced in C-order by default. This behavior is controlled by the `order` parameter.

New in version 1.9.0.

Parameters

order

[{'C', 'F', 'A'}, optional] Controls the memory layout of the bytes object. 'C' means C-order, 'F' means F-order, 'A' (short for *Any*) means 'F' if *a* is Fortran contiguous, 'C' otherwise. Default is 'C'.

Returns

s
[bytes] Python bytes exhibiting a copy of *a*'s raw data.

See also

frombuffer

Inverse of this operation, construct a 1-dimensional array from Python bytes.

Examples

```

>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'

```

tofile

`IsErrorArray.tofile(fid, sep="", format='%s')`

Write array to a file as text or binary (default).

Data is always written in ‘C’ order, independent of the order of *a*. The data produced by this method can be recovered using the function `fromfile()`.

Parameters

fid

[file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

sep

[str] Separator between array items for text output. If “” (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format

[str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When *fid* is a file object, array contents are directly written to the file, bypassing the file object’s `write` method. As a result, `tofile` cannot be used with files objects supporting compression (e.g., `GzipFile`) or file-like objects that do not support `fileno()` (e.g., `BytesIO`).

tolist

`IsErrorArray.tolist()`

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the `~numpy.ndarray.item` function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

Parameters

none

Returns

`y`
[object, or list of object, or list of list of object, or ...] The possibly nested list of array elements.

Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

tostring

`IsErrorArray.tostring(order='C')`

A compatibility alias for *tobytes*, with exactly the same behavior.

Despite its name, it returns *bytes* not *strs*.

Deprecated since version 1.19.0.

trace

`IsErrorArray.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Return the sum along diagonals of the array.

Refer to *numpy.trace* for full documentation.

See Also

`numpy.trace` : equivalent function

transpose

`IsErrorArray.transpose(*axes)`

Returns a view of the array with axes transposed.

Refer to *numpy.transpose* for full documentation.

Parameters

axes : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means that the array's *i*-th axis becomes the transposed array's *j*-th axis.
- *n* ints: same as an *n*-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form).

Returns

p
[ndarray] View of the array with its axes suitably permuted.

See Also

`transpose` : Equivalent function. `ndarray.T` : Array property returning the array transposed. `ndarray.reshape` : Give a new shape to an array without changing its data.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
>>> a.transpose()
array([1, 2, 3, 4])
```

var

`IsErrorArray.var(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True)`

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See Also

`numpy.var` : equivalent function

view

`IsErrorArray.view([dtype][, type])`

New view of array with the same data.

Note: Passing `None` for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

Parameters

`dtype`

[data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., `float32` or `int16`. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

`type`

[Python type, optional] Type of the returned view, e.g., `ndarray` or `matrix`. Again, omission of the parameter results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the last axis of *a* must be contiguous. This axis will be resized in the result.

Changed in version 1.23.0: Only the last axis needs to be contiguous. Previously, the entire array had to be C-contiguous.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2), (3, 4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1, 2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2., 3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0, 1] = 20
>>> x
array([(1, 20), (3, 4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int16)
>>> y = x[:, ::2]
>>> y
array([[1, 3],
       [4, 6]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
...
ValueError: To change to a dtype of a different size, the last axis must be
↳ contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[1, 3],
       [4, 6]], dtype=[('width', '<i2'), ('length', '<i2')])
```

However, views that change dtype are totally fine for arrays with a contiguous last axis, even if the rest of the axes are not C-contiguous:

```
>>> x = np.arange(2 * 3 * 4, dtype=np.int8).reshape(2, 3, 4)
>>> x.transpose(1, 0, 2).view(np.int16)
array([[256, 770],
       [3340, 3854]],
      [[1284, 1798],
       [4368, 4882]],
```

(continues on next page)

(continued from previous page)

```
[[2312, 2826],
 [5396, 5910]], dtype=int16)
```

`__init__()`

Attributes

<code>T</code>	View of the transposed array.
<code>base</code>	Base object if memory is from some other object.
<code>ctypes</code>	An object to simplify the interaction of the array with the ctypes module.
<code>data</code>	Python buffer object pointing to the start of the array's data.
<code>dtype</code>	Data-type of the array's elements.
<code>flags</code>	Information about the memory layout of the array.
<code>flat</code>	A 1-D iterator over the array.
<code>imag</code>	The imaginary part of the array.
<code>itemsize</code>	Length of one array element in bytes.
<code>nbytes</code>	Total bytes consumed by the elements of the array.
<code>ndim</code>	Number of array dimensions.
<code>real</code>	The real part of the array.
<code>shape</code>	Tuple of array dimensions.
<code>size</code>	Number of elements in the array.
<code>strides</code>	Tuple of bytes to step in each dimension when traversing an array.

T

`IsErrorArray.T`

View of the transposed array.

Same as `self.transpose()`.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.T
array([[1, 3],
       [2, 4]])
```

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
```

(continues on next page)

(continued from previous page)

```
>>> a.T  
array([1, 2, 3, 4])
```

See Also

`transpose`

base

`IsErrorArray.base`

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is `None`:

```
>>> x = np.array([1,2,3,4])  
>>> x.base is None  
True
```

Slicing creates a view, whose memory is shared with `x`:

```
>>> y = x[2:]  
>>> y.base is x  
True
```

ctypes

`IsErrorArray.ctypes`

An object to simplify the interaction of the array with the `ctypes` module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the `ctypes` module. The returned object has, among others, `data`, `shape`, and `strides` attributes (see Notes below) which themselves return `ctypes` objects that can be used as arguments to a shared library.

Parameters

`None`

Returns

c

[Python object] Possessing attributes data, shape, strides, etc.

See Also

`numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

`_ctypes.data`

A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

`_ctypes.shape`

(`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the C-integer corresponding to `dtype('p')` on this platform (see `~numpy.ctypeslib.c_intp`). This base-type could be `ctypes.c_int`, `ctypes.c_long`, or `ctypes.c_longlong` depending on the platform. The ctypes array contains the shape of the underlying array.

`_ctypes.strides`

(`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

`_ctypes.data_as(obj)`

Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

`_ctypes.shape_as(obj)`

Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

`_ctypes.strides_as(obj)`

Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the `ctypes` module is not available, then the `ctypes` attribute of array objects still returns something useful, but `ctypes` objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the `data` attribute.

Examples

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

data

`IsErrorArray.data`

Python buffer object pointing to the start of the array's data.

dtype

`IsErrorArray.dtype`

Data-type of the array's elements.

Warning: Setting `arr.dtype` is discouraged and may be deprecated in the future. Setting will replace the `dtype` without modifying the memory (see also `ndarray.view` and `ndarray.astype`).

Parameters

None

Returns

d : numpy dtype object

See Also

ndarray.astype : Cast the values contained in the array to a new data-type. ndarray.view : Create a view of the same data but a different data-type. numpy.dtype

Examples

```

>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>

```

flags

IsErrorArray.flags

Information about the memory layout of the array.

Attributes

C_CONTIGUOUS (C)

The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F)

The data is in a single, Fortran-style contiguous segment.

OWNDATA (O)

The array owns the memory it uses or borrows it from another object.

WRITEABLE (W)

The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

ALIGNED (A)

The data and all elements are aligned appropriately for the hardware.

WRITEBACKIFCOPY (X)

This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

FNC

F_CONTIGUOUS and not C_CONTIGUOUS.

FORC

F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B)

ALIGNED and WRITEABLE.

CARRAY (CA)

BEHAVED and C_CONTIGUOUS.

FARRAY (FA)

BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the `WRITEBACKIFCOPY`, `WRITEABLE`, and `ALIGNED` flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- `WRITEBACKIFCOPY` can only be set `False`.
- `ALIGNED` can only be set `True` if the data is truly aligned.
- `WRITEABLE` can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

flat

IsErrorArray.flat

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See Also

`flatten` : Return a copy of the array collapsed into one dimension.

`flatiter`

Examples

```

>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>

```

An assignment example:

```

>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])

```

imag

`IsErrorArray.imag`

The imaginary part of the array.

Examples

```

>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.          ,  0.70710678])
>>> x.imag.dtype
dtype('float64')

```

itemsize

`IsErrorArray.itemsize`

Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

nbytes

`IsErrorArray.nbytes`

Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

See Also

`sys.getsizeof`

Memory consumed by the object itself without parents in case view. This does include memory consumed by non-element attributes.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim

IsErrorArray.ndim

Number of array dimensions.

Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

real

IsErrorArray.real

The real part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.          ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

See Also

`numpy.real` : equivalent function

shape

IsErrorArray.shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with *numpy.reshape*, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

Warning: Setting `arr.shape` is discouraged and may be deprecated in the future. Using *ndarray.reshape* is the preferred approach.

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[:,2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

See Also

`numpy.shape` : Equivalent getter function. `numpy.reshape` : Function similar to setting `shape`. `ndarray.reshape` : Method similar to setting `shape`.

size

`IsErrorArray.size`

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Notes

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

strides

IsErrorArray.strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element ($i[0]$, $i[1]$, ..., $i[n]$) in an array a is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

Warning: Setting `arr.strides` is discouraged and may be deprecated in the future. `numpy.lib.stride_tricks.as_strided` should be preferred to create a new view of the same data in a safer way.

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array x will be (20, 4).

See Also

`numpy.lib.stride_tricks.as_strided`

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

IsNaArray

class IsNaArray

Methods

<code>__init__</code>	
<code>all</code>	Returns True if all elements evaluate to True.
<code>any</code>	Returns True if any of the elements of <i>a</i> evaluate to True.
<code>argmax</code>	Return indices of the maximum values along the given axis.
<code>argmin</code>	Return indices of the minimum values along the given axis.
<code>argpartition</code>	Returns the indices that would partition this array.
<code>argsort</code>	Returns the indices that would sort this array.
<code>astype</code>	Copy of the array, cast to a specified type.
<code>byteswap</code>	Swap the bytes of the array elements
<code>choose</code>	Use an index array to construct a new array from a set of choices.

continues on next page

Table 4 – continued from previous page

clip	Return an array whose values are limited to [min, max].
collapse	
compress	Return selected slices of this array along given axis.
conj	Complex-conjugate all elements.
conjugate	Return the complex conjugate, element-wise.
copy	Return a copy of the array.
cumprod	Return the cumulative product of the elements along the given axis.
cumsum	Return the cumulative sum of the elements along the given axis.
diagonal	Return specified diagonals.
dot	
dump	Dump a pickle of the array to the specified file.
dumps	Returns the pickle of the array as a string.
fill	Fill the array with a scalar value.
flatten	Return a copy of the array collapsed into one dimension.
getfield	Returns a field of the given array as a certain type.
item	Copy an element of an array to a standard Python scalar and return it.
itemset	Insert scalar into an array (scalar is cast to array's dtype, if possible)
max	Return the maximum along a given axis.
mean	Returns the average of the array elements along given axis.
min	Return the minimum along a given axis.
newbyteorder	Return the array with the same data viewed with a different byte order.
nonzero	Return the indices of the elements that are non-zero.
partition	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
prod	Return the product of the array elements over the given axis
ptp	Peak to peak (maximum - minimum) value along a given axis.
put	Set <code>a.flat[n] = values[n]</code> for all <code>n</code> in indices.
ravel	Return a flattened array.
repeat	Repeat elements of an array.
reshape	Returns an array containing the same data with a new shape.
resize	Change shape and size of array in-place.
round	Return <code>a</code> with each element rounded to the given number of decimals.
searchsorted	Find indices where elements of <code>v</code> should be inserted in <code>a</code> to maintain order.
setfield	Put a value into a specified place in a field defined by a data-type.

continues on next page

Table 4 – continued from previous page

<code>setflags</code>	Set array flags <code>WRITEABLE</code> , <code>ALIGNED</code> , <code>WRITEBACKIFCOPY</code> , respectively.
<code>sort</code>	Sort an array in-place.
<code>squeeze</code>	Remove axes of length one from <i>a</i> .
<code>std</code>	Returns the standard deviation of the array elements along given axis.
<code>sum</code>	Return the sum of the array elements over the given axis.
<code>swapaxes</code>	Return a view of the array with <i>axis1</i> and <i>axis2</i> interchanged.
<code>take</code>	Return an array formed from the elements of <i>a</i> at the given indices.
<code>tobytes</code>	Construct Python bytes containing the raw data bytes in the array.
<code>tofile</code>	Write array to a file as text or binary (default).
<code>tolist</code>	Return the array as an <code>a.ndim</code> -levels deep nested list of Python scalars.
<code>tostring</code>	A compatibility alias for <i>tobytes</i> , with exactly the same behavior.
<code>trace</code>	Return the sum along diagonals of the array.
<code>transpose</code>	Returns a view of the array with axes transposed.
<code>var</code>	Returns the variance of the array elements, along given axis.
<code>view</code>	New view of array with the same data.

`__init__`

`IsNaArray.__init__()`

`all`

`IsNaArray.all(axis=None, out=None, keepdims=False, *, where=True)`

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

See Also

`numpy.all` : equivalent function

any

`IsNaArray.any(axis=None, out=None, keepdims=False, *, where=True)`

Returns True if any of the elements of *a* evaluate to True.

Refer to `numpy.any` for full documentation.

See Also

`numpy.any` : equivalent function

argmax

`IsNaArray.argmax(axis=None, out=None, *, keepdims=False)`

Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

See Also

`numpy.argmax` : equivalent function

argmin

`IsNaArray.argmin(axis=None, out=None, *, keepdims=False)`

Return indices of the minimum values along the given axis.

Refer to `numpy.argmin` for detailed documentation.

See Also

`numpy.argmin` : equivalent function

argpartition

`IsNaArray.argpartition(kth, axis=-1, kind='introselect', order=None)`

Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

See Also

`numpy.argpartition` : equivalent function

argsort

`IsNaArray.argsort(axis=-1, kind=None, order=None)`

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See Also

`numpy.argsort` : equivalent function

astype

`IsNaArray.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)`

Copy of the array, cast to a specified type.

Parameters

dtype

[str or dtype] Typecode or data-type to which the array is cast.

order

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok

[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy

[bool, optional] By default, `astype` always returns a newly allocated array. If this is set to false, and the `dtype`, `order`, and `subok` requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t

[ndarray] Unless *copy* is False and the other conditions for returning the input array are satisfied (see description for *copy* input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by *dtype*, *order*.

Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for “unsafe” casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in ‘safe’ casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

Raises

ComplexWarning

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

byteswap

`IsNaArray.byteswap(inplace=False)`

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

Parameters

inplace

[bool, optional] If True, swap bytes in-place, default is False.

Returns

out

[ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([ 256,      1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='<S3')
```

`A.newbyteorder().byteswap()` produces an array with the same values
but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

choose

`IsNaArray.choose(choices, out=None, mode='raise')`

Use an index array to construct a new array from a set of choices.

Refer to *numpy.choose* for full documentation.

See Also

`numpy.choose` : equivalent function

clip

`IsNaArray.clip(min=None, max=None, out=None, **kwargs)`

Return an array whose values are limited to `[min, max]`. One of `max` or `min` must be given.

Refer to `numpy.clip` for full documentation.

See Also

`numpy.clip` : equivalent function

collapse

`IsNaArray.collapse(shape)`

compress

`IsNaArray.compress(condition, axis=None, out=None)`

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also

`numpy.compress` : equivalent function

conj

`IsNaArray.conj()`

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See Also

`numpy.conjugate` : equivalent function

conjugate

`IsNaArray.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

See Also

`numpy.conjugate` : equivalent function

copy

`IsNaArray.copy(order='C')`

Return a copy of the array.

Parameters

order

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy()` are very similar but have different default values for their `order=` arguments, and this function always passes sub-classes through.)

See also

`numpy.copy` : Similar function with different default behavior `numpy.copyto`

Notes

This function is the preferred method for creating an array copy. The function `numpy.copy()` is similar, but it defaults to using order 'K', and will not pass sub-classes through by default.

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y  
array([[1, 2, 3],  
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']  
True
```

cumprod

IsNaArray.**cumprod**(axis=None, dtype=None, out=None)

Return the cumulative product of the elements along the given axis.

Refer to *numpy.cumprod* for full documentation.

See Also

numpy.cumprod : equivalent function

cumsum

IsNaArray.**cumsum**(axis=None, dtype=None, out=None)

Return the cumulative sum of the elements along the given axis.

Refer to *numpy.cumsum* for full documentation.

See Also

numpy.cumsum : equivalent function

diagonal

IsNaArray.**diagonal**(offset=0, axis1=0, axis2=1)

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to *numpy.diagonal()* for full documentation.

See Also

numpy.diagonal : equivalent function

dot

`IsNaArray.dot()`

dump

`IsNaArray.dump(file)`

Dump a pickle of the array to the specified file. The array can be read back with `pickle.load` or `numpy.load`.

Parameters**file**

[str or Path] A string naming the dump file.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

dumps

`IsNaArray.dumps()`

Returns the pickle of the array as a string. `pickle.loads` will convert the string back to an array.

Parameters

None

fill

`IsNaArray.fill(value)`

Fill the array with a scalar value.

Parameters**value**

[scalar] All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.])
```


Fill expects a scalar value and always behaves the same as assigning to a single array element. The following is a rare example where this distinction is important:

```
>>> a = np.array([None, None], dtype=object)
>>> a[0] = np.array(3)
>>> a
array([array(3), None], dtype=object)
>>> a.fill(np.array(3))
>>> a
array([array(3), array(3)], dtype=object)
```

Where other forms of assignments will unpack the array being assigned:

```
>>> a[...] = np.array(3)
>>> a
array([3, 3], dtype=object)
```

flatten

IsNaArray.**flatten**(*order='C'*)

Return a copy of the array collapsed into one dimension.

Parameters

order

[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns

y

[ndarray] A copy of the input array, flattened to one dimension.

See Also

ravel : Return a flattened array. flat : A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

getfield

`IsNaArray.getfield(dtype, offset=0)`

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype `complex128` has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype

[str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset

[int] Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

item

`IsNaArray.item(*args)`

Copy an element of an array to a standard Python scalar and return it.

Parameters

*args : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z

[Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to *a[args]*, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

itemset

`IsNaArray.itemset(*args)`

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

***args**

[Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

max

`IsNaArray.max(axis=None, out=None, keepdims=False, initial=<no value>, where=True)`

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

See Also

`numpy.amax` : equivalent function

mean

`IsNaArray.mean(axis=None, dtype=None, out=None, keepdims=False, *, where=True)`

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See Also

`numpy.mean` : equivalent function

min

`IsNaArray.min(axis=None, out=None, keepdims=False, initial=<no value>, where=True)`

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See Also

`numpy.amin` : equivalent function

newbyteorder

`IsNaArray.newbyteorder(new_order='S', /)`

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters

`new_order`

[string, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- ‘S’ - swap dtype from current to opposite endian
- {‘<’, ‘little’} - little endian
- {‘>’, ‘big’} - big endian
- {‘=’, ‘native’} - native order, equivalent to `sys.byteorder`

- {'I', 'T'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order.

Returns

new_arr

[array] New array object with the dtype reflecting given change to the byte order.

nonzero

`IsNaArray.nonzero()`

Return the indices of the elements that are non-zero.

Refer to *numpy.nonzero* for full documentation.

See Also

`numpy.nonzero` : equivalent function

partition

`IsNaArray.partition(kth, axis=-1, kind='introselect', order=None)`

Rearranges the elements in the array in such a way that the value of the element in *kth* position is in the position it would be in a sorted array. All elements smaller than the *kth* element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

Parameters

kth

[int or sequence of ints] Element index to partition by. The *kth* element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of *kth* it will partition all elements indexed by *kth* of them into their sorted position at once.

Deprecated since version 1.22.0: Passing booleans as index is deprecated.

axis

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind

[{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

order

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

`numpy.partition` : Return a partitioned copy of an array. `argsort` : Indirect partition. `sort` : Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

prod

`IsNaArray.prod(axis=None, dtype=None, out=None, keepdims=False, initial=1, where=True)`

Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See Also

`numpy.prod` : equivalent function

ptp

`IsNaArray.ptp(axis=None, out=None, keepdims=False)`

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See Also

`numpy.ptp` : equivalent function

put

`IsNaArray.put(indices, values, mode='raise')`

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *numpy.put* for full documentation.

See Also

`numpy.put` : equivalent function

ravel

`IsNaArray.ravel([order])`

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

See Also

`numpy.ravel` : equivalent function

`ndarray.flat` : a flat iterator on the array.

repeat

`IsNaArray.repeat(repeats, axis=None)`

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

See Also

`numpy.repeat` : equivalent function

reshape

`IsNaArray.reshape(shape, order='C')`

Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

See Also

`numpy.reshape` : equivalent function

Notes

Unlike the free function *numpy.reshape*, this method on *ndarray* allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

resize

`IsNaArray.resize(new_shape, refcheck=True)`

Change shape and size of array in-place.

Parameters

new_shape

[tuple of ints, or *n* ints] Shape of resized array.

refcheck

[bool, optional] If False, reference count will not be checked. Default is True.

Returns

None

Raises

ValueError

If *a* does not own its own data or references or views to it exist, and the data memory must be changed.
PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

SystemError

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See Also

`resize` : Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and re-shaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

round

`IsNaArray.round(decimals=0, out=None)`

Return *a* with each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

See Also

`numpy.around` : equivalent function

searchsorted

`IsNaArray.searchsorted(v, side='left', sorter=None)`

Find indices where elements of *v* should be inserted in *a* to maintain order.

For full documentation, see `numpy.searchsorted`

See Also

`numpy.searchsorted` : equivalent function

setfield

`IsNaArray.setfield(val, dtype, offset=0)`

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters

val

[object] Value to be placed in field.

dtype

[dtype object] Data-type of the field in which to place *val*.

offset

[int, optional] The number of bytes into the field at which to place *val*.

Returns

None

See Also

`getfield`

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

setflags

`IsNaArray.setflags(write=None, align=None, uic=None)`

Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write

[bool, optional] Describes whether or not *a* can be written to.

align

[bool, optional] Describes whether or not *a* is aligned properly for its type.

uic

[bool, optional] Describes whether or not *a* is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: `WRITEBACKIFCOPY`, `WRITEABLE`, and `ALIGNED`.

`WRITEABLE` (W) the data area can be written to;

`ALIGNED` (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

`WRITEBACKIFCOPY` (X) this array is a copy of some other array (referenced by `.base`). When the C-API function `PyArray_ResolveWritebackIfCopy` is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
WRITEBACKIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

sort

`IsNaArray.sort(axis=-1, kind=None, order=None)`

Sort an array in-place. Refer to *numpy.sort* for full documentation.

Parameters

axis

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0: The 'stable' option was added.

order

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

`numpy.sort` : Return a sorted copy of an array. `numpy.argsort` : Indirect sort. `numpy.lexsort` : Indirect stable sort on multiple keys. `numpy.searchsorted` : Find elements in sorted array. `numpy.partition`: Partial sort.

Notes

See *numpy.sort* for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

squeeze

`IsNaArray.squeeze(axis=None)`

Remove axes of length one from *a*.

Refer to `numpy.squeeze` for full documentation.

See Also

`numpy.squeeze` : equivalent function

std

`IsNaArray.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True)`

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See Also

`numpy.std` : equivalent function

sum

`IsNaArray.sum(axis=None, dtype=None, out=None, keepdims=False, initial=0, where=True)`

Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See Also

`numpy.sum` : equivalent function

swapaxes

`IsNaArray.swapaxes(axis1, axis2)`

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also

`numpy.swapaxes` : equivalent function

take

`IsNaArray.take(indices, axis=None, out=None, mode='raise')`

Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

See Also

`numpy.take` : equivalent function

tobytes

`IsNaArray.tobytes(order='C')`

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object is produced in C-order by default. This behavior is controlled by the `order` parameter.

New in version 1.9.0.

Parameters

order

[['C', 'F', 'A'], optional] Controls the memory layout of the bytes object. 'C' means C-order, 'F' means F-order, 'A' (short for *Any*) means 'F' if *a* is Fortran contiguous, 'C' otherwise. Default is 'C'.

Returns

s

[bytes] Python bytes exhibiting a copy of *a*'s raw data.

See also

frombuffer

Inverse of this operation, construct a 1-dimensional array from Python bytes.

Examples

```

>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'

```

tofile

`IsNaArray.tofile(fid, sep="", format='%s')`

Write array to a file as text or binary (default).

Data is always written in ‘C’ order, independent of the order of *a*. The data produced by this method can be recovered using the function `fromfile()`.

Parameters

fid

[file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

sep

[str] Separator between array items for text output. If “” (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format

[str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When *fid* is a file object, array contents are directly written to the file, bypassing the file object’s `write` method. As a result, `tofile` cannot be used with files objects supporting compression (e.g., `GzipFile`) or file-like objects that do not support `fileno()` (e.g., `BytesIO`).

tolist

IsNaArray.tolist()

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the `~numpy.ndarray.item` function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

Parameters

none

Returns

`y`
[object, or list of object, or list of list of object, or ...] The possibly nested list of array elements.

Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

tostring

`IsNaArray.tostring(order='C')`

A compatibility alias for *tobytes*, with exactly the same behavior.

Despite its name, it returns *bytes* not *strs*.

Deprecated since version 1.19.0.

trace

`IsNaArray.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Return the sum along diagonals of the array.

Refer to *numpy.trace* for full documentation.

See Also

`numpy.trace` : equivalent function

transpose

`IsNaArray.transpose(*axes)`

Returns a view of the array with axes transposed.

Refer to *numpy.transpose* for full documentation.

Parameters

`axes` : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means that the array's *i*-th axis becomes the transposed array's *j*-th axis.
- *n* ints: same as an *n*-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form).

Returns

p
[ndarray] View of the array with its axes suitably permuted.

See Also

`transpose` : Equivalent function. `ndarray.T` : Array property returning the array transposed. `ndarray.reshape` : Give a new shape to an array without changing its data.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
>>> a.transpose()
array([1, 2, 3, 4])
```

var

`IsNaArray.var(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True)`

Returns the variance of the array elements, along given axis.

Refer to *numpy.var* for full documentation.

See Also

`numpy.var` : equivalent function

view

`IsNaArray.view([dtype][, type])`

New view of array with the same data.

Note: Passing `None` for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

Parameters

`dtype`

[data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., `float32` or `int16`. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

`type`

[Python type, optional] Type of the returned view, e.g., `ndarray` or `matrix`. Again, omission of the parameter results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the last axis of *a* must be contiguous. This axis will be resized in the result.

Changed in version 1.23.0: Only the last axis needs to be contiguous. Previously, the entire array had to be C-contiguous.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2), (3, 4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1, 2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2., 3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0, 1] = 20
>>> x
array([(1, 20), (3, 4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int16)
>>> y = x[:, ::2]
>>> y
array([[1, 3],
       [4, 6]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
...
ValueError: To change to a dtype of a different size, the last axis must be
↳ contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[1, 3],
       [4, 6]], dtype=[('width', '<i2'), ('length', '<i2')])
```

However, views that change dtype are totally fine for arrays with a contiguous last axis, even if the rest of the axes are not C-contiguous:

```
>>> x = np.arange(2 * 3 * 4, dtype=np.int8).reshape(2, 3, 4)
>>> x.transpose(1, 0, 2).view(np.int16)
array([[256, 770],
       [3340, 3854]],
      [[1284, 1798],
       [4368, 4882]],
```

(continues on next page)

(continued from previous page)

```
[[2312, 2826],
 [5396, 5910]], dtype=int16)
```

`__init__()`

Attributes

<code>T</code>	View of the transposed array.
<code>base</code>	Base object if memory is from some other object.
<code>ctypes</code>	An object to simplify the interaction of the array with the ctypes module.
<code>data</code>	Python buffer object pointing to the start of the array's data.
<code>dtype</code>	Data-type of the array's elements.
<code>flags</code>	Information about the memory layout of the array.
<code>flat</code>	A 1-D iterator over the array.
<code>imag</code>	The imaginary part of the array.
<code>itemsize</code>	Length of one array element in bytes.
<code>nbytes</code>	Total bytes consumed by the elements of the array.
<code>ndim</code>	Number of array dimensions.
<code>real</code>	The real part of the array.
<code>shape</code>	Tuple of array dimensions.
<code>size</code>	Number of elements in the array.
<code>strides</code>	Tuple of bytes to step in each dimension when traversing an array.

T

`IsNaArray.T`

View of the transposed array.

Same as `self.transpose()`.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.T
array([[1, 3],
       [2, 4]])
```

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
```

(continues on next page)

(continued from previous page)

```
>>> a.T  
array([1, 2, 3, 4])
```

See Also

`transpose`

base

`IsNaArray.base`

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is `None`:

```
>>> x = np.array([1,2,3,4])  
>>> x.base is None  
True
```

Slicing creates a view, whose memory is shared with `x`:

```
>>> y = x[2:]  
>>> y.base is x  
True
```

ctypes

`IsNaArray.ctypes`

An object to simplify the interaction of the array with the `ctypes` module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the `ctypes` module. The returned object has, among others, `data`, `shape`, and `strides` attributes (see Notes below) which themselves return `ctypes` objects that can be used as arguments to a shared library.

Parameters

`None`

Returns

c

[Python object] Possessing attributes data, shape, strides, etc.

See Also

numpy.ctypeslib

Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

`_ctypes.data`

A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

`_ctypes.shape`

(`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the C-integer corresponding to `dtype('p')` on this platform (see `~numpy.ctypeslib.c_intp`). This base-type could be `ctypes.c_int`, `ctypes.c_long`, or `ctypes.c_longlong` depending on the platform. The ctypes array contains the shape of the underlying array.

`_ctypes.strides`

(`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

`_ctypes.data_as(obj)`

Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

`_ctypes.shape_as(obj)`

Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

`_ctypes.strides_as(obj)`

Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the `ctypes` module is not available, then the `ctypes` attribute of array objects still returns something useful, but `ctypes` objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the `data` attribute.

Examples

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

data

`IsNaArray.data`

Python buffer object pointing to the start of the array's data.

dtype

`IsNaArray.dtype`

Data-type of the array's elements.

Warning: Setting `arr.dtype` is discouraged and may be deprecated in the future. Setting will replace the `dtype` without modifying the memory (see also `ndarray.view` and `ndarray.astype`).

Parameters

None

Returns

d : numpy dtype object

See Also

ndarray.astype : Cast the values contained in the array to a new data-type. ndarray.view : Create a view of the same data but a different data-type. numpy.dtype

Examples

```

>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>

```

flags

IsNaArray.flags

Information about the memory layout of the array.

Attributes

C_CONTIGUOUS (C)

The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F)

The data is in a single, Fortran-style contiguous segment.

OWNDATA (O)

The array owns the memory it uses or borrows it from another object.

WRITEABLE (W)

The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

ALIGNED (A)

The data and all elements are aligned appropriately for the hardware.

WRITEBACKIFCOPY (X)

This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

FNC

F_CONTIGUOUS and not C_CONTIGUOUS.

FORC

F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B)

ALIGNED and WRITEABLE.

CARRAY (CA)

BEHAVED and C_CONTIGUOUS.

FARRAY (FA)

BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the `WRITEBACKIFCOPY`, `WRITEABLE`, and `ALIGNED` flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- `WRITEBACKIFCOPY` can only be set `False`.
- `ALIGNED` can only be set `True` if the data is truly aligned.
- `WRITEABLE` can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

flat

IsNaArray.flat

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See Also

`flatten` : Return a copy of the array collapsed into one dimension.

`flatiter`

Examples

```

>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>

```

An assignment example:

```

>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])

```

imag

`IsNaArray.imag`

The imaginary part of the array.

Examples

```

>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.          ,  0.70710678])
>>> x.imag.dtype
dtype('float64')

```

itemsize

`IsNaArray.itemsize`

Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

nbytes

`IsNaArray.nbytes`

Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

See Also

`sys.getsizeof`

Memory consumed by the object itself without parents in case view. This does include memory consumed by non-element attributes.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim

IsNaArray.ndim

Number of array dimensions.

Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

real

IsNaArray.real

The real part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.          ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

See Also

`numpy.real` : equivalent function

shape

IsNaArray.shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with *numpy.reshape*, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

Warning: Setting `arr.shape` is discouraged and may be deprecated in the future. Using *ndarray.reshape* is the preferred approach.

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[:,2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

See Also

`numpy.shape` : Equivalent getter function. `numpy.reshape` : Function similar to setting `shape`. `ndarray.reshape` : Method similar to setting `shape`.

size

`IsNaArray.size`

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Notes

`a.size` returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

strides

IsNaArray.strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element ($i[0]$, $i[1]$, ..., $i[n]$) in an array a is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

Warning: Setting `arr.strides` is discouraged and may be deprecated in the future. `numpy.lib.stride_tricks.as_strided` should be preferred to create a new view of the same data in a safer way.

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array x will be (20, 4).

See Also

`numpy.lib.stride_tricks.as_strided`

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

IsNumberArray

class IsNumberArray

Methods

<code>__init__</code>	
<code>all</code>	Returns True if all elements evaluate to True.
<code>any</code>	Returns True if any of the elements of <i>a</i> evaluate to True.
<code>argmax</code>	Return indices of the maximum values along the given axis.
<code>argmin</code>	Return indices of the minimum values along the given axis.
<code>argpartition</code>	Returns the indices that would partition this array.
<code>argsort</code>	Returns the indices that would sort this array.
<code>astype</code>	Copy of the array, cast to a specified type.
<code>byteswap</code>	Swap the bytes of the array elements
<code>choose</code>	Use an index array to construct a new array from a set of choices.

continues on next page

Table 5 – continued from previous page

clip	Return an array whose values are limited to [min, max].
collapse	
compress	Return selected slices of this array along given axis.
conj	Complex-conjugate all elements.
conjugate	Return the complex conjugate, element-wise.
copy	Return a copy of the array.
cumprod	Return the cumulative product of the elements along the given axis.
cumsum	Return the cumulative sum of the elements along the given axis.
diagonal	Return specified diagonals.
dot	
dump	Dump a pickle of the array to the specified file.
dumps	Returns the pickle of the array as a string.
fill	Fill the array with a scalar value.
flatten	Return a copy of the array collapsed into one dimension.
getfield	Returns a field of the given array as a certain type.
item	Copy an element of an array to a standard Python scalar and return it.
itemset	Insert scalar into an array (scalar is cast to array's dtype, if possible)
max	Return the maximum along a given axis.
mean	Returns the average of the array elements along given axis.
min	Return the minimum along a given axis.
newbyteorder	Return the array with the same data viewed with a different byte order.
nonzero	Return the indices of the elements that are non-zero.
partition	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
prod	Return the product of the array elements over the given axis
ptp	Peak to peak (maximum - minimum) value along a given axis.
put	Set <code>a.flat[n] = values[n]</code> for all <code>n</code> in indices.
ravel	Return a flattened array.
repeat	Repeat elements of an array.
reshape	Returns an array containing the same data with a new shape.
resize	Change shape and size of array in-place.
round	Return <code>a</code> with each element rounded to the given number of decimals.
searchsorted	Find indices where elements of <code>v</code> should be inserted in <code>a</code> to maintain order.
setfield	Put a value into a specified place in a field defined by a data-type.

continues on next page

Table 5 – continued from previous page

<code>setflags</code>	Set array flags WRITEABLE, ALIGNED, WRITE-BACKIFCOPY, respectively.
<code>sort</code>	Sort an array in-place.
<code>squeeze</code>	Remove axes of length one from <i>a</i> .
<code>std</code>	Returns the standard deviation of the array elements along given axis.
<code>sum</code>	Return the sum of the array elements over the given axis.
<code>swapaxes</code>	Return a view of the array with <i>axis1</i> and <i>axis2</i> interchanged.
<code>take</code>	Return an array formed from the elements of <i>a</i> at the given indices.
<code>tobytes</code>	Construct Python bytes containing the raw data bytes in the array.
<code>tofile</code>	Write array to a file as text or binary (default).
<code>tolist</code>	Return the array as an <code>a.ndim</code> -levels deep nested list of Python scalars.
<code>tostring</code>	A compatibility alias for <i>tobytes</i> , with exactly the same behavior.
<code>trace</code>	Return the sum along diagonals of the array.
<code>transpose</code>	Returns a view of the array with axes transposed.
<code>var</code>	Returns the variance of the array elements, along given axis.
<code>view</code>	New view of array with the same data.

`__init__`

`IsNumberArray.__init__()`

`all`

`IsNumberArray.all(axis=None, out=None, keepdims=False, *, where=True)`

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

See Also

`numpy.all` : equivalent function

any

`IsNumberArray.any(axis=None, out=None, keepdims=False, *, where=True)`

Returns True if any of the elements of *a* evaluate to True.

Refer to `numpy.any` for full documentation.

See Also

`numpy.any` : equivalent function

argmax

`IsNumberArray.argmax(axis=None, out=None, *, keepdims=False)`

Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

See Also

`numpy.argmax` : equivalent function

argmin

`IsNumberArray.argmin(axis=None, out=None, *, keepdims=False)`

Return indices of the minimum values along the given axis.

Refer to `numpy.argmin` for detailed documentation.

See Also

`numpy.argmin` : equivalent function

argpartition

`IsNumberArray.argpartition(kth, axis=-1, kind='introselect', order=None)`

Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

See Also

`numpy.argpartition` : equivalent function

argsort

`IsNumberArray.argsort(axis=-1, kind=None, order=None)`

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See Also

`numpy.argsort` : equivalent function

astype

`IsNumberArray.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)`

Copy of the array, cast to a specified type.

Parameters

dtype

[str or dtype] Typecode or data-type to which the array is cast.

order

[['C', 'F', 'A', 'K'], optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting

[['no', 'equiv', 'safe', 'same_kind', 'unsafe'], optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok

[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy

[bool, optional] By default, `astype` always returns a newly allocated array. If this is set to false, and the `dtype`, `order`, and `subok` requirements are satisfied, the input array is returned instead of a copy.

Returns

arr_t

[ndarray] Unless *copy* is False and the other conditions for returning the input array are satisfied (see description for *copy* input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by *dtype*, *order*.

Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for “unsafe” casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in ‘safe’ casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

Raises

ComplexWarning

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

byteswap

`IsNumberArray.byteswap(inplace=False)`

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

Parameters

inplace

[bool, optional] If True, swap bytes in-place, default is False.

Returns

out

[ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([ 256,      1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='<S3')
```

A.newbyteorder().byteswap() produces an array with the same values
but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

choose

`IsNumberArray.choose(choices, out=None, mode='raise')`

Use an index array to construct a new array from a set of choices.

Refer to *numpy.choose* for full documentation.

See Also

`numpy.choose` : equivalent function

clip

`IsNumberArray.clip(min=None, max=None, out=None, **kwargs)`

Return an array whose values are limited to `[min, max]`. One of `max` or `min` must be given.

Refer to `numpy.clip` for full documentation.

See Also

`numpy.clip` : equivalent function

collapse

`IsNumberArray.collapse(shape)`

compress

`IsNumberArray.compress(condition, axis=None, out=None)`

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

See Also

`numpy.compress` : equivalent function

conj

`IsNumberArray.conj()`

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

See Also

`numpy.conjugate` : equivalent function

conjugate

`IsNumberArray.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

See Also

`numpy.conjugate` : equivalent function

copy

`IsNumberArray.copy(order='C')`

Return a copy of the array.

Parameters

order

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy()` are very similar but have different default values for their `order=` arguments, and this function always passes sub-classes through.)

See also

`numpy.copy` : Similar function with different default behavior `numpy.copyto`

Notes

This function is the preferred method for creating an array copy. The function `numpy.copy()` is similar, but it defaults to using order 'K', and will not pass sub-classes through by default.

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y  
array([[1, 2, 3],  
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']  
True
```

cumprod

`IsNumberArray.cumprod(axis=None, dtype=None, out=None)`

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See Also

`numpy.cumprod` : equivalent function

cumsum

`IsNumberArray.cumsum(axis=None, dtype=None, out=None)`

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See Also

`numpy.cumsum` : equivalent function

diagonal

`IsNumberArray.diagonal(offset=0, axis1=0, axis2=1)`

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal()` for full documentation.

See Also

`numpy.diagonal` : equivalent function

dot

`IsNumberArray.dot()`

dump

`IsNumberArray.dump(file)`

Dump a pickle of the array to the specified file. The array can be read back with `pickle.load` or `numpy.load`.

Parameters

file

[str or Path] A string naming the dump file.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

dumps

`IsNumberArray.dumps()`

Returns the pickle of the array as a string. `pickle.loads` will convert the string back to an array.

Parameters

None

fill

`IsNumberArray.fill(value)`

Fill the array with a scalar value.

Parameters

value

[scalar] All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.]
```

Fill expects a scalar value and always behaves the same as assigning to a single array element. The following is a rare example where this distinction is important:

```
>>> a = np.array([None, None], dtype=object)
>>> a[0] = np.array(3)
>>> a
array([array(3), None], dtype=object)
>>> a.fill(np.array(3))
>>> a
array([array(3), array(3)], dtype=object)
```

Where other forms of assignments will unpack the array being assigned:

```
>>> a[...] = np.array(3)
>>> a
array([3, 3], dtype=object)
```

flatten

`IsNumberArray.flatten(order='C')`

Return a copy of the array collapsed into one dimension.

Parameters

order

[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns

y

[ndarray] A copy of the input array, flattened to one dimension.

See Also

`ravel` : Return a flattened array. `flat` : A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

getfield

`IsNumberArray.getfield(dtype, offset=0)`

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype `complex128` has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype

[str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset

[int] Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

item

`IsNumberArray.item(*args)`

Copy an element of an array to a standard Python scalar and return it.

Parameters

*args : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z

[Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to *a[args]*, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

itemset

`IsNumberArray.itemset(*args)`

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

***args**

[Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

max

`IsNumberArray.max(axis=None, out=None, keepdims=False, initial=<no value>, where=True)`

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

See Also

`numpy.amax` : equivalent function

mean

`IsNumberArray.mean(axis=None, dtype=None, out=None, keepdims=False, *, where=True)`

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

See Also

`numpy.mean` : equivalent function

min

`IsNumberArray.min(axis=None, out=None, keepdims=False, initial=<no value>, where=True)`

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

See Also

`numpy.amin` : equivalent function

newbyteorder

`IsNumberArray.newbyteorder(new_order='S', /)`

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters

`new_order`

[string, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- ‘S’ - swap dtype from current to opposite endian
- {‘<’, ‘little’} - little endian
- {‘>’, ‘big’} - big endian
- {‘=’, ‘native’} - native order, equivalent to `sys.byteorder`

- {'I', 'T'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order.

Returns

new_arr

[array] New array object with the dtype reflecting given change to the byte order.

nonzero

`IsNumberArray.nonzero()`

Return the indices of the elements that are non-zero.

Refer to *numpy.nonzero* for full documentation.

See Also

`numpy.nonzero` : equivalent function

partition

`IsNumberArray.partition(kth, axis=-1, kind='introselect', order=None)`

Rearranges the elements in the array in such a way that the value of the element in *kth* position is in the position it would be in a sorted array. All elements smaller than the *kth* element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

Parameters

kth

[int or sequence of ints] Element index to partition by. The *kth* element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of *kth* it will partition all elements indexed by *kth* of them into their sorted position at once.

Deprecated since version 1.22.0: Passing booleans as index is deprecated.

axis

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind

[{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

order

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

`numpy.partition` : Return a partitioned copy of an array. `argsort` : Indirect partition. `sort` : Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

prod

`IsNumberArray.prod(axis=None, dtype=None, out=None, keepdims=False, initial=1, where=True)`

Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See Also

`numpy.prod` : equivalent function

ptp

`IsNumberArray.ptp(axis=None, out=None, keepdims=False)`

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See Also

`numpy.ptp` : equivalent function

put

`IsNumberArray.put(indices, values, mode='raise')`

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *numpy.put* for full documentation.

See Also

`numpy.put` : equivalent function

ravel

`IsNumberArray.ravel([order])`

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

See Also

`numpy.ravel` : equivalent function

`ndarray.flat` : a flat iterator on the array.

repeat

`IsNumberArray.repeat(repeats, axis=None)`

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

See Also

`numpy.repeat` : equivalent function

reshape

`IsNumberArray.reshape(shape, order='C')`

Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

See Also

`numpy.reshape` : equivalent function

Notes

Unlike the free function *numpy.reshape*, this method on *ndarray* allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

resize

`IsNumberArray.resize(new_shape, refcheck=True)`

Change shape and size of array in-place.

Parameters

new_shape

[tuple of ints, or *n* ints] Shape of resized array.

refcheck

[bool, optional] If False, reference count will not be checked. Default is True.

Returns

None

Raises

ValueError

If *a* does not own its own data or references or views to it exist, and the data memory must be changed.
PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

SystemError

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See Also

`resize` : Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and re-shaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

round

`IsNumberArray.round(decimals=0, out=None)`

Return *a* with each element rounded to the given number of decimals.

Refer to *numpy.around* for full documentation.

See Also

`numpy.around` : equivalent function

searchsorted

`IsNumberArray.searchsorted(v, side='left', sorter=None)`

Find indices where elements of *v* should be inserted in *a* to maintain order.

For full documentation, see *numpy.searchsorted*

See Also

`numpy.searchsorted` : equivalent function

setfield

`IsNumberArray.setfield(val, dtype, offset=0)`

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters

val

[object] Value to be placed in field.

dtype

[dtype object] Data-type of the field in which to place *val*.

offset

[int, optional] The number of bytes into the field at which to place *val*.

Returns

None

See Also

`getfield`

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

setflags

`IsNumberArray.setflags(write=None, align=None, uic=None)`

Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write

[bool, optional] Describes whether or not *a* can be written to.

align

[bool, optional] Describes whether or not *a* is aligned properly for its type.

uic

[bool, optional] Describes whether or not *a* is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: `WRITEBACKIFCOPY`, `WRITEABLE`, and `ALIGNED`.

`WRITEABLE` (W) the data area can be written to;

`ALIGNED` (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

`WRITEBACKIFCOPY` (X) this array is a copy of some other array (referenced by `.base`). When the C-API function `PyArray_ResolveWritebackIfCopy` is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
WRITEBACKIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

sort

`IsNumberArray.sort(axis=-1, kind=None, order=None)`

Sort an array in-place. Refer to *numpy.sort* for full documentation.

Parameters

axis

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0: The 'stable' option was added.

order

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

`numpy.sort` : Return a sorted copy of an array. `numpy.argsort` : Indirect sort. `numpy.lexsort` : Indirect stable sort on multiple keys. `numpy.searchsorted` : Find elements in sorted array. `numpy.partition`: Partial sort.

Notes

See *numpy.sort* for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([(b'a', 2), (b'c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

squeeze

`IsNumberArray.squeeze(axis=None)`

Remove axes of length one from *a*.

Refer to *numpy.squeeze* for full documentation.

See Also

`numpy.squeeze` : equivalent function

std

`IsNumberArray.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True)`

Returns the standard deviation of the array elements along given axis.

Refer to *numpy.std* for full documentation.

See Also

`numpy.std` : equivalent function

sum

`IsNumberArray.sum(axis=None, dtype=None, out=None, keepdims=False, initial=0, where=True)`

Return the sum of the array elements over the given axis.

Refer to *numpy.sum* for full documentation.

See Also

`numpy.sum` : equivalent function

swapaxes

`IsNumberArray.swapaxes(axis1, axis2)`

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also

`numpy.swapaxes` : equivalent function

take

`IsNumberArray.take(indices, axis=None, out=None, mode='raise')`

Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

See Also

`numpy.take` : equivalent function

tobytes

`IsNumberArray.tobytes(order='C')`

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object is produced in C-order by default. This behavior is controlled by the `order` parameter.

New in version 1.9.0.

Parameters

order

[{'C', 'F', 'A'}, optional] Controls the memory layout of the bytes object. 'C' means C-order, 'F' means F-order, 'A' (short for *Any*) means 'F' if *a* is Fortran contiguous, 'C' otherwise. Default is 'C'.

Returns

s

[bytes] Python bytes exhibiting a copy of *a*'s raw data.

See also

frombuffer

Inverse of this operation, construct a 1-dimensional array from Python bytes.

Examples

```

>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'

```

tofile

`IsNumberArray.tofile(fid, sep="", format='%s')`

Write array to a file as text or binary (default).

Data is always written in ‘C’ order, independent of the order of *a*. The data produced by this method can be recovered using the function `fromfile()`.

Parameters

fid

[file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

sep

[str] Separator between array items for text output. If “” (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format

[str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When *fid* is a file object, array contents are directly written to the file, bypassing the file object’s `write` method. As a result, `tofile` cannot be used with files objects supporting compression (e.g., `GzipFile`) or file-like objects that do not support `fileno()` (e.g., `BytesIO`).

tolist

IsNumberArray.tolist()

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the `~numpy.ndarray.item` function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

Parameters

none

Returns

y
[object, or list of object, or list of list of object, or ...] The possibly nested list of array elements.

Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

tostring

`IsNumberArray.tostring(order='C')`

A compatibility alias for *tobytes*, with exactly the same behavior.

Despite its name, it returns *bytes* not *strs*.

Deprecated since version 1.19.0.

trace

`IsNumberArray.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Return the sum along diagonals of the array.

Refer to *numpy.trace* for full documentation.

See Also

`numpy.trace` : equivalent function

transpose

`IsNumberArray.transpose(*axes)`

Returns a view of the array with axes transposed.

Refer to *numpy.transpose* for full documentation.

Parameters

axes : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means that the array's *i*-th axis becomes the transposed array's *j*-th axis.
- *n* ints: same as an *n*-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form).

Returns

p
[ndarray] View of the array with its axes suitably permuted.

See Also

`transpose` : Equivalent function. `ndarray.T` : Array property returning the array transposed. `ndarray.reshape` : Give a new shape to an array without changing its data.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
>>> a.transpose()
array([1, 2, 3, 4])
```

var

`IsNumberArray.var(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True)`

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See Also

`numpy.var` : equivalent function

view

`IsNumberArray.view([dtype][, type])`

New view of array with the same data.

Note: Passing `None` for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

Parameters

`dtype`

[data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., `float32` or `int16`. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

`type`

[Python type, optional] Type of the returned view, e.g., `ndarray` or `matrix`. Again, omission of the parameter results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the last axis of *a* must be contiguous. This axis will be resized in the result.

Changed in version 1.23.0: Only the last axis needs to be contiguous. Previously, the entire array had to be C-contiguous.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2), (3, 4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1, 2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2., 3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0, 1] = 20
>>> x
array([(1, 20), (3, 4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int16)
>>> y = x[:, ::2]
>>> y
array([[1, 3],
       [4, 6]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
...
ValueError: To change to a dtype of a different size, the last axis must be
↳ contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[1, 3],
       [4, 6]], dtype=[('width', '<i2'), ('length', '<i2')])
```

However, views that change dtype are totally fine for arrays with a contiguous last axis, even if the rest of the axes are not C-contiguous:

```
>>> x = np.arange(2 * 3 * 4, dtype=np.int8).reshape(2, 3, 4)
>>> x.transpose(1, 0, 2).view(np.int16)
array([[256, 770],
       [3340, 3854]],
      [[1284, 1798],
       [4368, 4882]],
```

(continues on next page)

(continued from previous page)

```
[[2312, 2826],
 [5396, 5910]], dtype=int16)
```

`__init__()`

Attributes

<code>T</code>	View of the transposed array.
<code>base</code>	Base object if memory is from some other object.
<code>ctypes</code>	An object to simplify the interaction of the array with the ctypes module.
<code>data</code>	Python buffer object pointing to the start of the array's data.
<code>dtype</code>	Data-type of the array's elements.
<code>flags</code>	Information about the memory layout of the array.
<code>flat</code>	A 1-D iterator over the array.
<code>imag</code>	The imaginary part of the array.
<code>itemsize</code>	Length of one array element in bytes.
<code>nbytes</code>	Total bytes consumed by the elements of the array.
<code>ndim</code>	Number of array dimensions.
<code>real</code>	The real part of the array.
<code>shape</code>	Tuple of array dimensions.
<code>size</code>	Number of elements in the array.
<code>strides</code>	Tuple of bytes to step in each dimension when traversing an array.

T

`IsNumberArray.T`

View of the transposed array.

Same as `self.transpose()`.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.T
array([[1, 3],
       [2, 4]])
```

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
```

(continues on next page)

(continued from previous page)

```
>>> a.T
array([1, 2, 3, 4])
```

See Also

`transpose`

base

`IsNumberArray.base`

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is `None`:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with `x`:

```
>>> y = x[2:]
>>> y.base is x
True
```

ctypes

`IsNumberArray.ctypes`

An object to simplify the interaction of the array with the `ctypes` module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the `ctypes` module. The returned object has, among others, `data`, `shape`, and `strides` attributes (see Notes below) which themselves return `ctypes` objects that can be used as arguments to a shared library.

Parameters

`None`

Returns

c

[Python object] Possessing attributes data, shape, strides, etc.

See Also

`numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

`_ctypes.data`

A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

`_ctypes.shape`

(`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the C-integer corresponding to `dtype('p')` on this platform (see `~numpy.ctypeslib.c_intp`). This base-type could be `ctypes.c_int`, `ctypes.c_long`, or `ctypes.c_longlong` depending on the platform. The ctypes array contains the shape of the underlying array.

`_ctypes.strides`

(`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

`_ctypes.data_as(obj)`

Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

`_ctypes.shape_as(obj)`

Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

`_ctypes.strides_as(obj)`

Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the `ctypes` module is not available, then the `ctypes` attribute of array objects still returns something useful, but `ctypes` objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the `data` attribute.

Examples

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

data

`IsNumberArray.data`

Python buffer object pointing to the start of the array's data.

dtype

`IsNumberArray.dtype`

Data-type of the array's elements.

Warning: Setting `arr.dtype` is discouraged and may be deprecated in the future. Setting will replace the `dtype` without modifying the memory (see also `ndarray.view` and `ndarray.astype`).

Parameters

None

Returns

d : numpy dtype object

See Also

ndarray.astype : Cast the values contained in the array to a new data-type. ndarray.view : Create a view of the same data but a different data-type. numpy.dtype

Examples

```

>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>

```

flags

IsNumberArray.flags

Information about the memory layout of the array.

Attributes

C_CONTIGUOUS (C)

The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F)

The data is in a single, Fortran-style contiguous segment.

OWNDATA (O)

The array owns the memory it uses or borrows it from another object.

WRITEABLE (W)

The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

ALIGNED (A)

The data and all elements are aligned appropriately for the hardware.

WRITEBACKIFCOPY (X)

This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

FNC

F_CONTIGUOUS and not C_CONTIGUOUS.

FORC

F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B)

ALIGNED and WRITEABLE.

CARRAY (CA)

BEHAVED and C_CONTIGUOUS.

FARRAY (FA)

BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the `WRITEBACKIFCOPY`, `WRITEABLE`, and `ALIGNED` flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- `WRITEBACKIFCOPY` can only be set `False`.
- `ALIGNED` can only be set `True` if the data is truly aligned.
- `WRITEABLE` can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

flat

IsNumberArray.flat

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See Also

`flatten` : Return a copy of the array collapsed into one dimension.

`flatiter`

Examples

```

>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>

```

An assignment example:

```

>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])

```

imag

`IsNumberArray.imag`

The imaginary part of the array.

Examples

```

>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.          ,  0.70710678])
>>> x.imag.dtype
dtype('float64')

```

itemsize

`IsNumberArray.itemsize`

Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

nbytes

`IsNumberArray.nbytes`

Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

See Also

`sys.getsizeof`

Memory consumed by the object itself without parents in case view. This does include memory consumed by non-element attributes.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim

`IsNumberArray.ndim`

Number of array dimensions.

Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

real

`IsNumberArray.real`

The real part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.          ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

See Also

`numpy.real` : equivalent function

shape

`IsNumberArray.shape`

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with *numpy.reshape*, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

Warning: Setting `arr.shape` is discouraged and may be deprecated in the future. Using *ndarray.reshape* is the preferred approach.

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[:,2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

See Also

`numpy.shape` : Equivalent getter function. `numpy.reshape` : Function similar to setting `shape`. `ndarray.reshape` : Method similar to setting `shape`.

size

`IsNumberArray.size`

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Notes

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

strides

IsNumberArray.strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element ($i[0]$, $i[1]$, ..., $i[n]$) in an array a is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

Warning: Setting `arr.strides` is discouraged and may be deprecated in the future. `numpy.lib.stride_tricks.as_strided` should be preferred to create a new view of the same data in a safer way.

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array x will be (20, 4).

See Also

`numpy.lib.stride_tricks.as_strided`

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

logic

Python equivalents of logical Excel functions.

Functions

solve_cycle

xand

xif

xiferror

xiferror_return

xifna

xifs

xswitch

solve_cycle**solve_cycle**(*args)**xand****xand**(logical, *logicals, func=<built-in method reduce of numpy.ufunc object>)**xif****xif**(condition, x=True, y=False)**xiferror****xiferror**(val, val_if_error)**xiferror_return****xiferror_return**(res, val, val_if_error)**xifna****xifna**(val, val_if_error)**xifs****xifs**(*cond_vals)**xswitch****xswitch**(val, *args)**look**

Python equivalents of lookup and reference Excel functions.

Functions

xaddress

xcolumn

xindex

xlookup

xmatch

xrow

xsingle

xaddress

xaddress(*row_num*, *column_num*, *abs_num=1*, *al=True*, *sheet_text=None*)

xcolumn

xcolumn(*cell=None*, *ref=None*)

xindex

xindex(*array*, *row_num*, *col_num=None*, *area_num=1*)

xlookup

xlookup(*lookup_val*, *lookup_vec*, *result_vec=None*, *match_type=1*)

xmatch

xmatch(*lookup_value*, *lookup_array*, *match_type=1*)

xrow

xrow(*cell=None, ref=None*)

xsingle

xsingle(*cell, rng*)

math

Python equivalents of math and trigonometry Excel functions.

Functions

<i>round_up</i>
<i>xarabic</i>
<i>xarctan2</i>
<i>xceiling</i>
<i>xceiling_math</i>
<i>xcot</i>
<i>xdecimal</i>
<i>xeven</i>
<i>xfact</i>
<i>xfactdouble</i>
<i>xgcd</i>
<i>xlcm</i>
<i>xmod</i>
<i>xmround</i>
<i>xodd</i>
<i>xpower</i>
<i>xrandbetween</i>
<i>xroman</i>
<i>xround</i>
<i>xsrqtpi</i>
<i>xsum</i>
<i>xsumproduct</i>
<i>xtrunc</i>

round_up**round_up**(*x*)**xarabic****xarabic**(*text*)**xarctan2****xarctan2**(*x*, *y*)**xceiling****xceiling**(*num*, *sig*, *ceil*=<built-in function ceil>, *dfl*=0)**xceiling_math****xceiling_math**(*num*, *sig*=None, *mode*=0, *ceil*=<built-in function ceil>)**xcot****xcot**(*x*, *func*=<ufunc 'tan'>)**xdecimal****xdecimal**(*text*, *radix*)**xeven****xeven**(*x*)**xfact****xfact**(*number*, *fact*=<built-in function factorial>, *limit*=0)

xfactdouble

xfactdouble(*number*)

xgcd

xgcd(**args*)

xlcm

xlcm(**args*)

xmod

xmod(*x*, *y*)

xmround

xmround(**args*)

xodd

xodd(*x*)

xpower

xpower(*number*, *power*)

xrandbetween

xrandbetween(*bottom*, *top*)

xroman

xroman(*num*, *form*=0)

xround

xround(*x*, *d*, *func*=<function round_up>)

xsrqtpi

xsrqtpi(*number*)

xsum

xsum(**args*, *func*=<function sum>)

xsumproduct

xsumproduct(**args*)

xtrunc

xtrunc(*x*, *d*=0, *func*=<built-in function trunc>)

operators

Python equivalents of Excel operators.

Functions

logic_input_parser

logic_input_parser

logic_input_parser(*x*, *y*)

stat

Python equivalents of statistical Excel functions.

Functions

<code>xcorrel</code>
<code>xforecast</code>
<code>xfunc</code>
<code>xslope</code>
<code>xsort</code>
<code>xstdev</code>

xcorrel

xcorrel(*arr1*, *arr2*)

xforecast

xforecast(*x*, *a*=None, *b*=None)

xfunc

xfunc(*args, *func*=<built-in function max>, *check*=<function is_number>, *convert*=None, *default*=0, *_raise*=True)

xslope

xslope(*yp*, *xp*)

xsort

xsort(*values*, *k*, *large*=True)

xstdev

xstdev(*args*, *ddof*=1, *func*=<function std>)

text

Python equivalents of text Excel functions.

Functions

<code>xconcat</code>
<code>xfind</code>
<code>xleft</code>
<code>xmid</code>
<code>xreplace</code>
<code>xright</code>
<code>xsearch</code>
<code>xtext</code>
<code>xvalue</code>

xconcat

xconcat(*text*, **args*)

xfind

xfind(*find_text*, *within_text*, *start_num*=1)

xleft

xleft(*from_str*, *num_chars*)

xmid

xmid(*from_str*, *start_num*, *num_chars*)

xreplace

xreplace(*old_text*, *start_num*, *num_chars*, *new_text*)

xright

xright(*from_str*, *num_chars*)

xsearch

xsearch(*find_text*, *within_text*, *start_num*=1)

xtext

xtext(*value*, *format_code*)

xvalue

xvalue(*value*)

Functions

<code>args2list</code>	
<code>args2vals</code>	
<code>clean_values</code>	
<code>convert2float</code>	
<code>convert_nan</code>	
<code>convert_noshp</code>	
<code>flatten</code>	
<code>get_error</code>	
<code>get_functions</code>	
<code>is_not_empty</code>	
<code>is_number</code>	
<code>not_implemented</code>	
<code>parse_ranges</code>	
<code>raise_errors</code>	
<code>replace_empty</code>	
<code>text2num</code>	
<code>to_number</code>	
<code>value_return</code>	
<code>wrap_func</code>	
<code>wrap_impure_func</code>	
<code>wrap_ranges_func</code>	
<code>wrap_ufunc</code>	Helps call a numpy universal function (ufunc).
<code>xfilter</code>	

args2list

args2list(*max_shape*, *shapes*, **args*)

args2vals

args2vals(*args*)

clean_values

clean_values(*values*)

convert2float

convert2float(**a*)

convert_nan

convert_nan(*value*, *default*=#NUM!)

convert_noshp

convert_noshp(*value*)

flatten

flatten(*v*, *check*=<function is_number>, *drop_empty*=False)

get_error

get_error(**vals*)

get_functions

get_functions()

is_not_empty**is_not_empty**(*v*)**is_number****is_number**(*number*, *xl_return=True*, *bool_return=False*)**not_implemented****not_implemented**(**args*, ***kwargs*)**parse_ranges****parse_ranges**(**args*, ***kw*)**raise_errors****raise_errors**(**args*)**replace_empty****replace_empty**(*x*, *empty=0*)**text2num****text2num**(**args*, ***kwargs*)**to_number****to_number**(**args*, ***kwargs*)**value_return****value_return**(*res*, **args*)

wrap_func

wrap_func(*func*, *ranges=False*)

wrap_impure_func

wrap_impure_func(*func*)

wrap_ranges_func

wrap_ranges_func(*func*, *n_out=1*)

wrap_ufunc

wrap_ufunc(*func*, *input_parser*=<function <lambda>>, *check_error*=<function get_error>, *args_parser*=<function <lambda>>, *otype*=<class 'formulas.functions.Array'>, *ranges=False*, *return_func*=<function <lambda>>, *check_nan=True*, ***kw*)

Helps call a numpy universal function (ufunc).

xfilter

xfilter(*accumulator*, *test_range*, *condition*, *operating_range=None*)

Classes

Array

Array

class Array

Methods

<code>__init__</code>	
<code>all</code>	Returns True if all elements evaluate to True.
<code>any</code>	Returns True if any of the elements of <i>a</i> evaluate to True.
<code>argmax</code>	Return indices of the maximum values along the given axis.

continues on next page

Table 6 – continued from previous page

<code>argmin</code>	Return indices of the minimum values along the given axis.
<code>argpartition</code>	Returns the indices that would partition this array.
<code>argsort</code>	Returns the indices that would sort this array.
<code>astype</code>	Copy of the array, cast to a specified type.
<code>byteswap</code>	Swap the bytes of the array elements
<code>choose</code>	Use an index array to construct a new array from a set of choices.
<code>clip</code>	Return an array whose values are limited to <code>[min, max]</code> .
<code>collapse</code>	
<code>compress</code>	Return selected slices of this array along given axis.
<code>conj</code>	Complex-conjugate all elements.
<code>conjugate</code>	Return the complex conjugate, element-wise.
<code>copy</code>	Return a copy of the array.
<code>cumprod</code>	Return the cumulative product of the elements along the given axis.
<code>cumsum</code>	Return the cumulative sum of the elements along the given axis.
<code>diagonal</code>	Return specified diagonals.
<code>dot</code>	
<code>dump</code>	Dump a pickle of the array to the specified file.
<code>dumps</code>	Returns the pickle of the array as a string.
<code>fill</code>	Fill the array with a scalar value.
<code>flatten</code>	Return a copy of the array collapsed into one dimension.
<code>getfield</code>	Returns a field of the given array as a certain type.
<code>item</code>	Copy an element of an array to a standard Python scalar and return it.
<code>itemset</code>	Insert scalar into an array (scalar is cast to array's dtype, if possible)
<code>max</code>	Return the maximum along a given axis.
<code>mean</code>	Returns the average of the array elements along given axis.
<code>min</code>	Return the minimum along a given axis.
<code>newbyteorder</code>	Return the array with the same data viewed with a different byte order.
<code>nonzero</code>	Return the indices of the elements that are non-zero.
<code>partition</code>	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
<code>prod</code>	Return the product of the array elements over the given axis
<code>ptp</code>	Peak to peak (maximum - minimum) value along a given axis.
<code>put</code>	Set <code>a.flat[n] = values[n]</code> for all <code>n</code> in indices.
<code>ravel</code>	Return a flattened array.
<code>repeat</code>	Repeat elements of an array.
<code>reshape</code>	Returns an array containing the same data with a new shape.

continues on next page

Table 6 – continued from previous page

<code>resize</code>	Change shape and size of array in-place.
<code>round</code>	Return <i>a</i> with each element rounded to the given number of decimals.
<code>searchsorted</code>	Find indices where elements of <i>v</i> should be inserted in <i>a</i> to maintain order.
<code>setfield</code>	Put a value into a specified place in a field defined by a data-type.
<code>setflags</code>	Set array flags WRITEABLE, ALIGNED, WRITE-BACKIFCOPY, respectively.
<code>sort</code>	Sort an array in-place.
<code>squeeze</code>	Remove axes of length one from <i>a</i> .
<code>std</code>	Returns the standard deviation of the array elements along given axis.
<code>sum</code>	Return the sum of the array elements over the given axis.
<code>swapaxes</code>	Return a view of the array with <i>axis1</i> and <i>axis2</i> interchanged.
<code>take</code>	Return an array formed from the elements of <i>a</i> at the given indices.
<code>tobytes</code>	Construct Python bytes containing the raw data bytes in the array.
<code>tofile</code>	Write array to a file as text or binary (default).
<code>tolist</code>	Return the array as an <i>a.ndim</i> -levels deep nested list of Python scalars.
<code>tostring</code>	A compatibility alias for <i>tobytes</i> , with exactly the same behavior.
<code>trace</code>	Return the sum along diagonals of the array.
<code>transpose</code>	Returns a view of the array with axes transposed.
<code>var</code>	Returns the variance of the array elements, along given axis.
<code>view</code>	New view of array with the same data.

`__init__`

`Array.__init__()`

all

Array.**all**(*axis=None, out=None, keepdims=False, *, where=True*)

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

See Also

numpy.all : equivalent function

any

Array.**any**(*axis=None, out=None, keepdims=False, *, where=True*)

Returns True if any of the elements of *a* evaluate to True.

Refer to *numpy.any* for full documentation.

See Also

numpy.any : equivalent function

argmax

Array.**argmax**(*axis=None, out=None, *, keepdims=False*)

Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

See Also

numpy.argmax : equivalent function

argmin

Array.**argmin**(*axis=None, out=None, *, keepdims=False*)

Return indices of the minimum values along the given axis.

Refer to *numpy.argmin* for detailed documentation.

See Also

`numpy.argmin` : equivalent function

argpartition

`Array.argpartition(kth, axis=-1, kind='introselect', order=None)`

Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

See Also

`numpy.argpartition` : equivalent function

argsort

`Array.argsort(axis=-1, kind=None, order=None)`

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See Also

`numpy.argsort` : equivalent function

astype

`Array.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)`

Copy of the array, cast to a specified type.

Parameters

dtype

[str or dtype] Typecode or data-type to which the array is cast.

order

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.

- ‘safe’ means only casts which can preserve values are allowed.
- ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
- ‘unsafe’ means any data conversions may be done.

subok

[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy

[bool, optional] By default, `astype` always returns a newly allocated array. If this is set to false, and the *dtype*, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

Returns**arr_t**

[ndarray] Unless *copy* is False and the other conditions for returning the input array are satisfied (see description for *copy* input parameter), *arr_t* is a new array of the same shape as the input array, with *dtype*, *order* given by *dtype*, *order*.

Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for “unsafe” casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in ‘safe’ casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

Raises**ComplexWarning**

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

byteswap

Array.**byteswap**(*inplace=False*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

Parameters

inplace

[bool, optional] If True, swap bytes in-place, default is False.

Returns

out

[ndarray] The byteswapped array. If *inplace* is True, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([ 256,      1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='|S3')
```

A.newbyteorder().byteswap() produces an array with the same values
but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

choose

Array.**choose**(*choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to *numpy.choose* for full documentation.

See Also

numpy.choose : equivalent function

clip

Array.**clip**(*min=None*, *max=None*, *out=None*, ***kwargs*)

Return an array whose values are limited to [*min*, *max*]. One of *max* or *min* must be given.

Refer to *numpy.clip* for full documentation.

See Also

numpy.clip : equivalent function

collapse

Array.**collapse**(*shape*)

compress

Array.**compress**(*condition*, *axis=None*, *out=None*)

Return selected slices of this array along given axis.

Refer to *numpy.compress* for full documentation.

See Also

numpy.compress : equivalent function

conj

Array.**conj**()

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

See Also

`numpy.conjugate` : equivalent function

conjugate

`Array.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

See Also

`numpy.conjugate` : equivalent function

copy

`Array.copy(order='C')`

Return a copy of the array.

Parameters

order

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy()` are very similar but have different default values for their `order=` arguments, and this function always passes sub-classes through.)

See also

`numpy.copy` : Similar function with different default behavior `numpy.copyto`

Notes

This function is the preferred method for creating an array copy. The function `numpy.copy()` is similar, but it defaults to using order 'K', and will not pass sub-classes through by default.

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

cumprod

Array.**cumprod**(*axis=None, dtype=None, out=None*)

Return the cumulative product of the elements along the given axis.

Refer to *numpy.cumprod* for full documentation.

See Also

numpy.cumprod : equivalent function

cumsum

Array.**cumsum**(*axis=None, dtype=None, out=None*)

Return the cumulative sum of the elements along the given axis.

Refer to *numpy.cumsum* for full documentation.

See Also

numpy.cumsum : equivalent function

diagonal

Array.**diagonal**(*offset=0, axis1=0, axis2=1*)

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to *numpy.diagonal()* for full documentation.

See Also

`numpy.diagonal` : equivalent function

dot

`Array.dot()`

dump

`Array.dump(file)`

Dump a pickle of the array to the specified file. The array can be read back with `pickle.load` or `numpy.load`.

Parameters

file

[str or Path] A string naming the dump file.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

dumps

`Array.dumps()`

Returns the pickle of the array as a string. `pickle.loads` will convert the string back to an array.

Parameters

None

fill

`Array.fill(value)`

Fill the array with a scalar value.

Parameters

value

[scalar] All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.])
```

Fill expects a scalar value and always behaves the same as assigning to a single array element. The following is a rare example where this distinction is important:

```
>>> a = np.array([None, None], dtype=object)
>>> a[0] = np.array(3)
>>> a
array([array(3), None], dtype=object)
>>> a.fill(np.array(3))
>>> a
array([array(3), array(3)], dtype=object)
```

Where other forms of assignments will unpack the array being assigned:

```
>>> a[...] = np.array(3)
>>> a
array([3, 3], dtype=object)
```

flatten

Array.**flatten**(*order='C'*)

Return a copy of the array collapsed into one dimension.

Parameters

order

[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

Returns

y
[ndarray] A copy of the input array, flattened to one dimension.

See Also

ravel : Return a flattened array. flat : A 1-D flat iterator over the array.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

getfield

Array.**getfield**(dtype, offset=0)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

dtype

[str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset

[int] Number of bytes to skip before beginning the element view.

Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:


```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

item

Array.item(*args)

Copy an element of an array to a standard Python scalar and return it.

Parameters

**args* : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

Returns

z

[Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to *a[args]*, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
```

(continues on next page)

(continued from previous page)

```

0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1

```

itemset

`Array.itemset(*args)`

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

**args*

[Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

Examples

```

>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])

```

max

Array.**max**(*axis=None, out=None, keepdims=False, initial=<no value>, where=True*)

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

See Also

numpy.amax : equivalent function

mean

Array.**mean**(*axis=None, dtype=None, out=None, keepdims=False, *, where=True*)

Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

See Also

numpy.mean : equivalent function

min

Array.**min**(*axis=None, out=None, keepdims=False, initial=<no value>, where=True*)

Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

See Also

numpy.amin : equivalent function

newbyteorder

Array.**newbyteorder**(*new_order='S', /*)

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters

new_order

[string, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'little'} - little endian
- {'>', 'big'} - big endian
- {'=', 'native'} - native order, equivalent to *sys.byteorder*
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order.

Returns

new_arr

[array] New array object with the dtype reflecting given change to the byte order.

nonzero

Array.nonzero()

Return the indices of the elements that are non-zero.

Refer to *numpy.nonzero* for full documentation.

See Also

`numpy.nonzero` : equivalent function

partition

Array.partition(*kth*, *axis*=-1, *kind*='introselect', *order*=None)

Rearranges the elements in the array in such a way that the value of the element in *kth* position is in the position it would be in a sorted array. All elements smaller than the *kth* element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

Parameters

kth

[int or sequence of ints] Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

Deprecated since version 1.22.0: Passing booleans as index is deprecated.

axis

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind

[{ 'introselect' }, optional] Selection algorithm. Default is 'introselect'.

order

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

`numpy.partition` : Return a partitioned copy of an array. `argsort` : Indirect partition. `sort` : Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

prod

`Array.prod(axis=None, dtype=None, out=None, keepdims=False, initial=1, where=True)`

Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See Also

`numpy.prod` : equivalent function

ptp

`Array.ptp(axis=None, out=None, keepdims=False)`

Peak to peak (maximum - minimum) value along a given axis.

Refer to *numpy.ptp* for full documentation.

See Also

`numpy.ptp` : equivalent function

put

`Array.put(indices, values, mode='raise')`

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *numpy.put* for full documentation.

See Also

`numpy.put` : equivalent function

ravel

`Array.ravel([order])`

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

See Also

`numpy.ravel` : equivalent function

`ndarray.flat` : a flat iterator on the array.

repeat

Array.**repeat**(*repeats*, *axis=None*)

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

See Also

numpy.repeat : equivalent function

reshape

Array.**reshape**(*shape*, *order='C'*)

Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

See Also

numpy.reshape : equivalent function

Notes

Unlike the free function *numpy.reshape*, this method on *ndarray* allows the elements of the shape parameter to be passed in as separate arguments. For example, *a.reshape(10, 11)* is equivalent to *a.reshape((10, 11))*.

resize

Array.**resize**(*new_shape*, *refcheck=True*)

Change shape and size of array in-place.

Parameters

new_shape

[tuple of ints, or *n* ints] Shape of resized array.

refcheck

[bool, optional] If False, reference count will not be checked. Default is True.

Returns

None

Raises

ValueError

If *a* does not own its own data or references or views to it exist, and the data memory must be changed.
PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

SystemError

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

See Also

`resize` : Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and re-shaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:


```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

round

Array.**round**(*decimals=0, out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to *numpy.around* for full documentation.

See Also

numpy.around : equivalent function

searchsorted

Array.**searchsorted**(*v, side='left', sorter=None*)

Find indices where elements of *v* should be inserted in *a* to maintain order.

For full documentation, see *numpy.searchsorted*

See Also

numpy.searchsorted : equivalent function

setfield

`Array.setfield(val, dtype, offset=0)`

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters

val

[object] Value to be placed in field.

dtype

[dtype object] Data-type of the field in which to place *val*.

offset

[int, optional] The number of bytes into the field at which to place *val*.

Returns

None

See Also

`getfield`

Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

setflags

`Array.setflags(write=None, align=None, uic=None)`

Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

write

[bool, optional] Describes whether or not *a* can be written to.

align

[bool, optional] Describes whether or not *a* is aligned properly for its type.

uic

[bool, optional] Describes whether or not *a* is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by `.base`). When the C-API function `PyArray_ResolveWritebackIfCopy` is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

Examples

```

>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True

```

(continues on next page)

(continued from previous page)

```

    ALIGNED : True
    WRITEBACKIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
    C_CONTIGUOUS : True
    F_CONTIGUOUS : False
    OWNDATA : True
    WRITEABLE : False
    ALIGNED : False
    WRITEBACKIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True

```

sort

`Array.sort(axis=-1, kind=None, order=None)`

Sort an array in-place. Refer to *numpy.sort* for full documentation.

Parameters

axis

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind

[['quicksort', 'mergesort', 'heapsort', 'stable'], optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0: The 'stable' option was added.

order

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See Also

`numpy.sort` : Return a sorted copy of an array. `numpy.argsort` : Indirect sort. `numpy.lexsort` : Indirect stable sort on multiple keys. `numpy.searchsorted` : Find elements in sorted array. `numpy.partition` : Partial sort.

Notes

See *numpy.sort* for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

squeeze

Array.squeeze(*axis=None*)

Remove axes of length one from *a*.

Refer to *numpy.squeeze* for full documentation.

See Also

numpy.squeeze : equivalent function

std

Array.std(*axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True*)

Returns the standard deviation of the array elements along given axis.

Refer to *numpy.std* for full documentation.

See Also

`numpy.std` : equivalent function

sum

`Array.sum(axis=None, dtype=None, out=None, keepdims=False, initial=0, where=True)`

Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See Also

`numpy.sum` : equivalent function

swapaxes

`Array.swapaxes(axis1, axis2)`

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

See Also

`numpy.swapaxes` : equivalent function

take

`Array.take(indices, axis=None, out=None, mode='raise')`

Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

See Also

`numpy.take` : equivalent function

tobytes

`Array.tobytes(order='C')`

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object is produced in C-order by default. This behavior is controlled by the `order` parameter.

New in version 1.9.0.

Parameters

order

[[‘C’, ‘F’, ‘A’], optional] Controls the memory layout of the bytes object. ‘C’ means C-order, ‘F’ means F-order, ‘A’ (short for *Any*) means ‘F’ if *a* is Fortran contiguous, ‘C’ otherwise. Default is ‘C’.

Returns

s

[bytes] Python bytes exhibiting a copy of *a*’s raw data.

See also

frombuffer

Inverse of this operation, construct a 1-dimensional array from Python bytes.

Examples

```
>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'
```

tofile

Array.tofile(*fid*, *sep*=", *format*='%s')

Write array to a file as text or binary (default).

Data is always written in ‘C’ order, independent of the order of *a*. The data produced by this method can be recovered using the function `fromfile()`.

Parameters

fid

[file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

sep

[str] Separator between array items for text output. If “” (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format

[str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When `fid` is a file object, array contents are directly written to the file, bypassing the file object's `write` method. As a result, `tofile` cannot be used with files objects supporting compression (e.g., `GzipFile`) or file-like objects that do not support `fileno()` (e.g., `BytesIO`).

tolist

`Array.tolist()`

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the `~numpy.ndarray.item` function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

Parameters

none

Returns

y
[object, or list of object, or list of list of object, or ...] The possibly nested list of array elements.

Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
```

(continues on next page)

(continued from previous page)

```
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

tostring

`Array.tolist(order='C')`

A compatibility alias for *tobytes*, with exactly the same behavior.

Despite its name, it returns *bytes* not *strs*.

Deprecated since version 1.19.0.

trace

`Array.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Return the sum along diagonals of the array.

Refer to *numpy.trace* for full documentation.

See Also

`numpy.trace` : equivalent function

transpose

Array.**transpose**(*axes)

Returns a view of the array with axes transposed.

Refer to *numpy.transpose* for full documentation.

Parameters

axes : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means that the array's *i*-th axis becomes the transposed array's *j*-th axis.
- *n* ints: same as an *n*-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form).

Returns

p

[ndarray] View of the array with its axes suitably permuted.

See Also

transpose : Equivalent function. ndarray.T : Array property returning the array transposed. ndarray.reshape : Give a new shape to an array without changing its data.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
>>> a.transpose()
array([1, 2, 3, 4])
```

var

`Array.var(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True)`

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

See Also

`numpy.var` : equivalent function

view

`Array.view([dtype][, type])`

New view of array with the same data.

Note: Passing `None` for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

Parameters**dtype**

[data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., `float32` or `int16`. Omitting it results in the view having the same data-type as `a`. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

type

[Python type, optional] Type of the returned view, e.g., `ndarray` or `matrix`. Again, omission of the parameter results in type preservation.

Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of `ndarray_subclass` that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the last axis of `a` must be contiguous. This axis will be resized in the result.

Changed in version 1.23.0: Only the last axis needs to be contiguous. Previously, the entire array had to be C-contiguous.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2), (3, 4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1, 2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2., 3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0, 1] = 20
>>> x
array([(1, 20), (3, 4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int16)
>>> y = x[:, ::2]
>>> y
array([[1, 3],
       [4, 6]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
...
ValueError: To change to a dtype of a different size, the last axis must be
↳ contiguous
>>> z = y.copy()
```

(continues on next page)

(continued from previous page)

```
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[1, 3],
       [(4, 6)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

However, views that change dtype are totally fine for arrays with a contiguous last axis, even if the rest of the axes are not C-contiguous:

```
>>> x = np.arange(2 * 3 * 4, dtype=np.int8).reshape(2, 3, 4)
>>> x.transpose(1, 0, 2).view(np.int16)
array([[[ 256,  770],
        [3340, 3854]],

       [[1284, 1798],
        [4368, 4882]],

       [[2312, 2826],
        [5396, 5910]]], dtype=int16)
```

`__init__()`

Attributes

<code>T</code>	View of the transposed array.
<code>base</code>	Base object if memory is from some other object.
<code>ctypes</code>	An object to simplify the interaction of the array with the ctypes module.
<code>data</code>	Python buffer object pointing to the start of the array's data.
<code>dtype</code>	Data-type of the array's elements.
<code>flags</code>	Information about the memory layout of the array.
<code>flat</code>	A 1-D iterator over the array.
<code>imag</code>	The imaginary part of the array.
<code>itemsize</code>	Length of one array element in bytes.
<code>nbytes</code>	Total bytes consumed by the elements of the array.
<code>ndim</code>	Number of array dimensions.
<code>real</code>	The real part of the array.
<code>shape</code>	Tuple of array dimensions.
<code>size</code>	Number of elements in the array.
<code>strides</code>	Tuple of bytes to step in each dimension when traversing an array.

T

Array.T

View of the transposed array.

Same as `self.transpose()`.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.T
array([[1, 3],
       [2, 4]])
```

```
>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
>>> a.T
array([1, 2, 3, 4])
```

See Also

[transpose](#)

base

Array.base

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is `None`:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with `x`:

```
>>> y = x[2:]
>>> y.base is x
True
```

ctypes

Array.**ctypes**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters

None

Returns

c
[Python object] Possessing attributes data, shape, strides, etc.

See Also

`numpy.ctypeslib`

Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

_ctypes.data

A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

_ctypes.shape

(`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the C-integer corresponding to `dtype('p')` on this platform (see `~numpy.ctypeslib.c_intp`). This base-type could be `ctypes.c_int`, `ctypes.c_long`, or `ctypes.c_longlong` depending on the platform. The ctypes array contains the shape of the underlying array.

_ctypes.strides

(`c_intp*self.ndim`): A ctypes array of length `self.ndim` where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

`_ctypes.data_as(obj)`

Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

`_ctypes.shape_as(obj)`

Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

`_ctypes.strides_as(obj)`

Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the data attribute.

Examples

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

data

Array.data

Python buffer object pointing to the start of the array's data.

dtype

Array.dtype

Data-type of the array's elements.

Warning: Setting `arr.dtype` is discouraged and may be deprecated in the future. Setting will replace the `dtype` without modifying the memory (see also `ndarray.view` and `ndarray.astype`).

Parameters

None

Returns

d : numpy dtype object

See Also

`ndarray.astype` : Cast the values contained in the array to a new data-type. `ndarray.view` : Create a view of the same data but a different data-type. `numpy.dtype`

Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

flags

Array.flags

Information about the memory layout of the array.

Attributes

C_CONTIGUOUS (C)

The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F)

The data is in a single, Fortran-style contiguous segment.

OWNDATA (O)

The array owns the memory it uses or borrows it from another object.

WRITEABLE (W)

The data area can be written to. Setting this to `False` locks the data, making it read-only. A view (slice, etc.) inherits `WRITEABLE` from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a `RuntimeError` exception.

ALIGNED (A)

The data and all elements are aligned appropriately for the hardware.

WRITEBACKIFCOPY (X)

This array is a copy of some other array. The C-API function `PyArray_ResolveWritebackIfCopy` must be called before deallocating to the base array will be updated with the contents of this array.

FNC

`F_CONTIGUOUS` and not `C_CONTIGUOUS`.

FORC

`F_CONTIGUOUS` or `C_CONTIGUOUS` (one-segment test).

BEHAVED (B)

`ALIGNED` and `WRITEABLE`.

CARRAY (CA)

`BEHAVED` and `C_CONTIGUOUS`.

FARRAY (FA)

`BEHAVED` and `F_CONTIGUOUS` and not `C_CONTIGUOUS`.

Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the `WRITEBACKIFCOPY`, `WRITEABLE`, and `ALIGNED` flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- `WRITEBACKIFCOPY` can only be set `False`.
- `ALIGNED` can only be set `True` if the data is truly aligned.
- `WRITEABLE` can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

flat

Array.flat

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See Also

`flatten` : Return a copy of the array collapsed into one dimension.

`flatiter`

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

imag

Array.imag

The imaginary part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.          ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

itemsize

Array.itemsize

Length of one array element in bytes.

Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

nbytes

Array.nbytes

Total bytes consumed by the elements of the array.

Notes

Does not include memory consumed by non-element attributes of the array object.

See Also

sys.getsizeof

Memory consumed by the object itself without parents in case view. This does include memory consumed by non-element attributes.

Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim

Array.ndim

Number of array dimensions.

Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

real

Array.real

The real part of the array.

Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.          ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

See Also

`numpy.real` : equivalent function

shape

Array.shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with *numpy.reshape*, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

Warning: Setting `arr.shape` is discouraged and may be deprecated in the future. Using *ndarray.reshape* is the preferred approach.

Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[:,2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

See Also

`numpy.shape` : Equivalent getter function. `numpy.reshape` : Function similar to setting `shape`. `ndarray.reshape` : Method similar to setting `shape`.

size

Array.size

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Notes

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

strides

Array.strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the “`ndarray.rst`” file in the NumPy reference guide.

Warning: Setting `arr.strides` is discouraged and may be deprecated in the future. `numpy.lib.stride_tricks.as_strided` should be preferred to create a new view of the same data in a safer way.

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be (20, 4).

See Also

`numpy.lib.stride_tricks.as_strided`

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
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]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

reshape(*shape*, *order*='C')

Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

See Also

`numpy.reshape` : equivalent function

Notes

Unlike the free function *numpy.reshape*, this method on *ndarray* allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

2.2.7.6 ranges

It provides Ranges class.

Classes

Ranges

Ranges

class `Ranges`(*ranges=()*, *values=None*)

Methods

`__init__`

`format_range`

`get_range`

`intersect`

`push`

`pushes`

`set_value`

`simplify`

`__init__`

`Ranges.__init__(ranges=(), values=None)`

`format_range`

`static Ranges.format_range(*args, **kwargs)`

`get_range`

`static Ranges.get_range(ref, context)`

`intersect`

`Ranges.intersect(other)`

`push`

`Ranges.push(ref, value=empty, context=None)`

`pushes`

`Ranges.pushes(refs, values=(), context=None)`

`set_value`

`Ranges.set_value(rng, value=empty)`

`simplify`

`Ranges.simplify()`

`__init__(ranges=(), values=None)`

Attributes

<code>ranges</code>
<code>values</code>
<code>is_set</code>
<code>value</code>

ranges

Ranges.**ranges**

values

Ranges.**values**

is_set

property Ranges.**is_set**

value

property Ranges.**value**

2.2.7.7 cell

It provides Cell class.

Functions

format_output

wrap_cell_func

format_output

format_output(*rng*, *value*)

wrap_cell_func

wrap_cell_func(*func*, *parse_args*=<function <lambda>>, *parse_kwargs*=<function <lambda>>)

Classes

<i>Cell</i>
<i>CellWrapper</i>
<i>RangesAssembler</i>
<i>Ref</i>

Cell

class `Cell`(*reference*, *value*, *context=None*, *check_formula=True*, *replace_missing_ref=True*)

Methods

<code>__init__</code>
<code>add</code>
<code>compile</code>
<code>update_inputs</code>

`__init__`

`Cell.__init__(reference, value, context=None, check_formula=True, replace_missing_ref=True)`

`add`

`Cell.add(dsp, context=None)`

`compile`

`Cell.compile(references=None, context=None)`

update_inputs

`Cell.update_inputs(references=None)`

`__init__(reference, value, context=None, check_formula=True, replace_missing_ref=True)`

Attributes

parser

parser

`Cell.parser = <formulas.parser.Parser object>`

CellWrapper

`class CellWrapper(func, parse_args, parse_kwargs)`

Methods

<code>__init__</code>
<code>check_cycles</code>

__init__

`CellWrapper.__init__(func, parse_args, parse_kwargs)`

check_cycles

`CellWrapper.check_cycles(cycle)`

`__init__(func, parse_args, parse_kwargs)`

RangesAssembler

class `RangesAssembler`(*ref*, *context=None*, *compact=1*)

Methods

<code>__init__</code>

<code>add</code>

<code>push</code>

`__init__`

`RangesAssembler.__init__(ref, context=None, compact=1)`

add

`RangesAssembler.add(dsp)`

push

`RangesAssembler.push(indices, output=None)`

`__init__(ref, context=None, compact=1)`

Attributes

<code>output</code>

output

property `RangesAssembler.output`

Ref

class **Ref**(*reference*, *value*, *context=None*, *check_formula=True*)

Methods

`__init__`

`add`

`compile`

`update_inputs`

`__init__`

Ref.**__init__**(*reference*, *value*, *context=None*, *check_formula=True*)

add

Ref.add(*dsp*, *context=None*)

compile

Ref.compile(*references=None*, *context=None*)

update_inputs

Ref.update_inputs(*references=None*)

`__init__`(*reference*, *value*, *context=None*, *check_formula=True*)

Attributes

`parser`

parser

Ref.`parser` = <`formulas.parser.Parser` object>

2.2.7.8 excel

It provides Excel model class.

Sub-Modules:

<code>cycle</code>	A dependency-free version of networkx's implementation of <i>simple_cycles</i> .
<code>xlreader</code>	It provides a custom Excel Reader class.

cycle

A dependency-free version of networkx's implementation of *simple_cycles*.

Functions

<code>simple_cycles</code>

simple_cycles

`simple_cycles(graph, copy=True)`

xlreader

It provides a custom Excel Reader class.

Functions

<code>load_workbook</code>

load_workbook

`load_workbook(filename, **kw)`

Classes

XlReader

XlReader

class `XlReader(*args, raw_date=True, **kwargs)`

Methods

`__init__`

`read`

`read_chartsheet`

`read_custom`

`read_manifest`

`read_properties`

`read_strings`

`read_theme`

`read_workbook`

`read_worksheets`

`__init__`

`XlReader.__init__(*args, raw_date=True, **kwargs)`

read

`XlReader.read()`

read_chartsheet

`XlReader.read_chartsheet(sheet, rel)`

read_custom

`XlReader.read_custom()`

read_manifest

`XlReader.read_manifest()`

read_properties

`XlReader.read_properties()`

read_strings

`XlReader.read_strings()`

read_theme

`XlReader.read_theme()`

read_workbook

`XlReader.read_workbook()`

read_worksheets

`XlReader.read_worksheets()`

`__init__(*args, raw_date=True, **kwargs)`

Classes

*ExcelModel**XlCircular*

ExcelModel

`class ExcelModel`

Methods

<code>__init__</code>
<code>add_book</code>
<code>add_cell</code>
<code>add_references</code>
<code>add_sheet</code>
<code>assemble</code>
<code>calculate</code>
<code>compare</code>
<code>compile</code>
<code>compile_cell</code>
<code>complete</code>
<code>external_links</code>
<code>finish</code>
<code>formula_ranges</code>
<code>formula_references</code>
<code>from_dict</code>
<code>from_ranges</code>
<code>inverse_references</code>
<code>load</code>
<code>loads</code>
<code>push</code>
<code>pushes</code>
<code>solve_circular</code>
<code>to_dict</code>
<code>write</code>

__init__

ExcelModel.**__init__**()

add_book

ExcelModel.**add_book**(*book=None, context=None, data_only=False*)

add_cell

ExcelModel.**add_cell**(*cell, context, formula_ranges*)

add_references

ExcelModel.**add_references**(*book, context=None*)

add_sheet

ExcelModel.**add_sheet**(*worksheet, context*)

assemble

ExcelModel.**assemble**(*compact=1*)

calculate

ExcelModel.**calculate**(*args, **kwargs)

compare

ExcelModel.**compare**(*fpaths, *solution=None, tolerance=1e-06, **kwargs*)

compile

ExcelModel.**compile**(*inputs, outputs*)

compile_cell

ExcelModel.**compile_cell**(*cell*, *context*, *references*, *formula_references*)

complete

ExcelModel.**complete**(*stack=None*)

external_links

ExcelModel.**external_links**(*ctx*)

finish

ExcelModel.**finish**(*complete=True*, *circular=False*, *assemble=True*)

formula_ranges

ExcelModel.**formula_ranges**(*ctx*)

formula_references

ExcelModel.**formula_references**(*ctx*)

from_dict

ExcelModel.**from_dict**(*adict*, *context=None*, *assemble=True*, *ref=True*)

from_ranges

ExcelModel.**from_ranges**(**ranges*)

inverse_references

ExcelModel.**inverse_references**()

load

`ExcelModel.load(filename)`

loads

`ExcelModel.loads(*file_names)`

push

`ExcelModel.push(worksheet, context)`

pushes

`ExcelModel.pushes(*worksheets, context=None)`

solve_circular

`ExcelModel.solve_circular()`

to_dict

`ExcelModel.to_dict()`

write

`ExcelModel.write(books=None, solution=None, dirpath=None)`

`__init__()`

Attributes

references

references

property ExcelModel.references

compile_class

alias of DispatchPipe

XlCircular

class XlCircular(*args)

Methods

<code>__init__</code>	
<code>capitalize</code>	Return a capitalized version of the string.
<code>casefold</code>	Return a version of the string suitable for caseless comparisons.
<code>center</code>	Return a centered string of length width.
<code>count</code>	Return the number of non-overlapping occurrences of substring sub in string S[start:end].
<code>encode</code>	Encode the string using the codec registered for encoding.
<code>endswith</code>	Return True if S ends with the specified suffix, False otherwise.
<code>expandtabs</code>	Return a copy where all tab characters are expanded using spaces.
<code>find</code>	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
<code>format</code>	Return a formatted version of S, using substitutions from args and kwargs.
<code>format_map</code>	Return a formatted version of S, using substitutions from mapping.
<code>index</code>	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
<code>isalnum</code>	Return True if the string is an alpha-numeric string, False otherwise.
<code>isalpha</code>	Return True if the string is an alphabetic string, False otherwise.
<code>isascii</code>	Return True if all characters in the string are ASCII, False otherwise.
<code>isdecimal</code>	Return True if the string is a decimal string, False otherwise.
<code>isdigit</code>	Return True if the string is a digit string, False otherwise.
<code>isidentifier</code>	Return True if the string is a valid Python identifier, False otherwise.
<code>islower</code>	Return True if the string is a lowercase string, False otherwise.

continues on next page

Table 7 – continued from previous page

<code>isnumeric</code>	Return True if the string is a numeric string, False otherwise.
<code>isprintable</code>	Return True if the string is printable, False otherwise.
<code>isspace</code>	Return True if the string is a whitespace string, False otherwise.
<code>istitle</code>	Return True if the string is a title-cased string, False otherwise.
<code>isupper</code>	Return True if the string is an uppercase string, False otherwise.
<code>join</code>	Concatenate any number of strings.
<code>ljust</code>	Return a left-justified string of length width.
<code>lower</code>	Return a copy of the string converted to lowercase.
<code>lstrip</code>	Return a copy of the string with leading whitespace removed.
<code>maketrans</code>	Return a translation table usable for <code>str.translate()</code> .
<code>partition</code>	Partition the string into three parts using the given separator.
<code>removeprefix</code>	Return a str with the given prefix string removed if present.
<code>removesuffix</code>	Return a str with the given suffix string removed if present.
<code>replace</code>	Return a copy with all occurrences of substring old replaced by new.
<code>rfind</code>	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
<code>rindex</code>	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
<code>rjust</code>	Return a right-justified string of length width.
<code>rpartition</code>	Partition the string into three parts using the given separator.
<code>rsplit</code>	Return a list of the substrings in the string, using sep as the separator string.
<code>rstrip</code>	Return a copy of the string with trailing whitespace removed.
<code>split</code>	Return a list of the substrings in the string, using sep as the separator string.
<code>splitlines</code>	Return a list of the lines in the string, breaking at line boundaries.
<code>startswith</code>	Return True if S starts with the specified prefix, False otherwise.
<code>strip</code>	Return a copy of the string with leading and trailing whitespace removed.
<code>swapcase</code>	Convert uppercase characters to lowercase and lowercase characters to uppercase.
<code>title</code>	Return a version of the string where each word is titlecased.
<code>translate</code>	Replace each character in the string using the given translation table.
<code>upper</code>	Return a copy of the string converted to uppercase.
<code>zfill</code>	Pad a numeric string with zeros on the left, to fill a field of the given width.

__init__

`XlCircular.__init__(*args)`

capitalize

`XlCircular.capitalize()`

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold

`XlCircular.casefold()`

Return a version of the string suitable for caseless comparisons.

center

`XlCircular.center(width, fillchar=' ', /)`

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count

`XlCircular.count(sub[, start[, end]]) → int`

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode

`XlCircular.encode(encoding='utf-8', errors='strict')`

Encode the string using the codec registered for encoding.

encoding

The encoding in which to encode the string.

errors

The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a `UnicodeEncodeError`. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with `codecs.register_error` that can handle `UnicodeEncodeErrors`.

endswith

`XlCircular.endswith(suffix[, start[, end]])` → bool

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs

`XlCircular.expandtabs(tabsize=8)`

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find

`XlCircular.find(sub[, start[, end]])` → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format

`XlCircular.format(*args, **kwargs)` → str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map

`XlCircular.format_map(mapping)` → str

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index

`XlCircular.index(sub[, start[, end]])` → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum

`XlCircular.isalnum()`

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha

`XlCircular.isalpha()`

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii

`XlCircular.isascii()`

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal

`XlCircular.isdecimal()`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit

`XlCircular.isdigit()`

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier

`XlCircular.isidentifier()`

Return True if the string is a valid Python identifier, False otherwise.

Call `keyword.iskeyword(s)` to test whether string `s` is a reserved identifier, such as “def” or “class”.

islower

XlCircular.islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric

XlCircular.isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable

XlCircular.isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace

XlCircular.isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle

XlCircular.istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper

XlCircular.isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join

`XlCircular.join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

ljust

`XlCircular.ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower

`XlCircular.lower()`

Return a copy of the string converted to lowercase.

lstrip

`XlCircular.lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

maketrans

`static XlCircular.maketrans()`

Return a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition

`XlCircular.partition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

removeprefix

`XlCircular.removeprefix(prefix, /)`

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return `string[len(prefix):]`. Otherwise, return a copy of the original string.

removesuffix

`XlCircular.removesuffix(suffix, /)`

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return `string[:-len(suffix)]`. Otherwise, return a copy of the original string.

replace

`XlCircular.replace(old, new, count=-1, /)`

Return a copy with all occurrences of substring `old` replaced by `new`.

count

Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument `count` is given, only the first `count` occurrences are replaced.

rfind

`XlCircular.rfind(sub[, start[, end]]) → int`

Return the highest index in `S` where substring `sub` is found, such that `sub` is contained within `S[start:end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

Return -1 on failure.

rindex

`XlCircular.rindex(sub[, start[, end]]) → int`

Return the highest index in `S` where substring `sub` is found, such that `sub` is contained within `S[start:end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

Raises `ValueError` when the substring is not found.

rjust

`XlCircular.rjust(width, fillchar=' ', /)`

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition

`XlCircular.rpartition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit

`XlCircular.rsplit(sep=None, maxsplit=-1)`

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including `n` `r` `t` `f` and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip

`XlCircular.rstrip(chars=None, /)`

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split

`XlCircular.split(sep=None, maxsplit=-1)`

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including `n` `r` `t` `f` and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, `str.split()` is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines

`XlCircular.splitlines(keepends=False)`

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless `keepends` is given and true.

startswith

`XlCircular.startswith(prefix[, start[, end]]) → bool`

Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. `prefix` can also be a tuple of strings to try.

strip

`XlCircular.strip(chars=None, /)`

Return a copy of the string with leading and trailing whitespace removed.

If `chars` is given and not None, remove characters in `chars` instead.

swapcase

`XlCircular.swapcase()`

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title

`XlCircular.title()`

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate

`XlCircular.translate(table, /)`

Replace each character in the string using the given translation table.

table

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper

`XlCircular.upper()`

Return a copy of the string converted to uppercase.

zfill

`XlCircular.zfill(width, /)`

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

`__init__(*args)`

2.2.8 Changelog

2.2.8.1 v1.2.6 (2023-11-15)

Feat

- (builder) [#104](#): Allow custom reference definition.
- (test): Update test cases.
- (operand) [#106](#): Accept number like .3 to be parsed.
- (text) [#113](#): Add *TEXT* function without fraction formatting.
- (logic): Update logic functions according to new excel logic.
- (text) [#113](#): Add *VALUE* function.
- (math) [#121](#): Improve performances of *SUMPRODUCT*, *PRODUCT*, *SUM*, and *SUMIF*.
- (setup): Update requirements.
- (core): Change development status.
- (core): Add support for python 3.10 and 3.11.
- (functions) [#121](#): Improve handling of *EMPTY* values.
- (excel): Avoid using *flatten* function in basic routines.
- (doc): Add Read the Docs configuration file.
- (excel): Add tolerance when comparing two excels.
- (excel): Add compare method to verify if formulas is able to replicate excel values.

Fix

- (doc): Remove broken badge.
- (excel) #100: Correct reading rounding from excel.
- (math) #100: Correct *TRUNC* defaults.
- (tokens) #113: Correct *sheet_id* definition.
- (functions): Correct dill pickling error.
- (excel): Correct reference parsing when loading from JSON.
- (functions): Use an alternative method of vectorize when more than 32 arguments are provided.
- (lookup): Correct *MATCH*, *LOOKUP*, *HLOOKUP*, and *VLOOKUP* behaviour when empty values are given.
- (date): Correct *DATEDIF* behaviour when unit is lowercase.
- (test): Use regex for unstable tests due to changes in last digits.
- (doc): Correct documentation bug due to new *sphinx*.
- (excel) #114: Update reading code according to *openpyxl* ≥ 3.1 .

2.2.8.2 v1.2.6 (2022-12-13)**Fix**

- (setup): Update *schedula* requirement.

2.2.8.3 v1.2.5 (2022-11-07)**Fix**

- (parser): Correct missing raise.
- (excel): Skip hidden named ranges.

2.2.8.4 v1.2.4 (2022-07-02)**Feat**

- (core): Improve speed performance.
- (cell): Improve speed *RangesAssembler* definition.

Fix

- (cell): Correct range assembler defaults when no *sheet_id* is defined.
- (math) #99: Convert args into np.arrays in func *xsumproduct*.
- (look): Correct lookup parser for float and strings.

2.2.8.5 v1.2.3 (2022-05-10)

Feat

- (test): Add more error logs.
- (test): Improve code coverage.
- (builder): Add *compile_class* attribute to *AstBuilder*.
- (info): Add *ISODD*, *ISEVEN*, *ISBLANK*, *ISTEXT*, *ISNONTEXT*, and *ISLOGICAL* functions.

Fix

- (excel): Correct file path excel definition.
- (logic): Correct *SWITCH* error handling.
- (actions): Rename workflow name.
- (readme): Correct badge link for dependencies status.
- (excel): Correct *basedir* reference to load files.
- (date): Correct *YEARFRAC* and *DATEDIF* formulation.
- (cell): Enable R1C1 notation for absolute and relative references.
- (cell): Correct RangeAssembler value assignment.

2.2.8.6 v1.2.2 (2022-01-22)

Fix

- (excel): Correct function compilation from excel.

2.2.8.7 v1.2.1 (2022-01-21)

Feat

- (functions): Improve performances caching results.
- (excel): Make replacing missing ref optional in *from_dict* method.
- (excel) #73, #75: Improve performances to parse full ranges.

Fix

- (excel): Correct compile function when inputs are computed with a default function.

2.2.8.8 v1.2.0 (2021-12-23)**Feat**

- (binder): Refresh environment binder for 2021.
- (look) #87: Add *ADDRESS* function.
- (test): Update test cases.
- (financial) #74, #87: Add *FV*, *PV*, *IPMT*, *PMT*, *PPMT*, *RATE*, *CUMIPMT*, and *NPER* functions.
- (info, logic): Add *ISNA* and *IFNA* functions.
- (date) #87: Add *WEEKDAY*, *WEEKNUM*, *ISOWEEKNUM*, and *DATEDIF* functions.
- (stat, math) #87: Add *SLOPE* and *PRODUCT* functions.
- (stats) #87: Add *CORREL* and *MEDIAN* functions.
- (bin): Add *bin* folder.
- (actions): Add test cases.
- (stats) #80: Add *FORECAST* and *FORECAST.LINEAR* functions.
- (excel) #82: Add inverse of simple references.

Fix

- (stat): Correct *LARGE* and *SMALL* error handling.
- (actions): Skip *Setup Graphviz* when not needed.
- (actions): Correct coverall setting.
- (actions): Remove unstable test case.
- (actions): Disable fail fast.
- (date, stat): Correct collapsed return value.
- (function) #78, #79, #91: Correct import error.

2.2.8.9 v1.1.1 (2021-10-13)**Feat**

- (excel): Improve performances of *complete* method.
- (setup): Add add python 3.9 in *setup.py*.
- (functions): Add *SEARCH*, *ISNUMBER*, and *EDATE* functions.
- (travis): Update python version for coveralls.

Fix

- (doc): Correct missing documentation link.
- (doc): Correct typo.
- (operator) #70: Correct % operator preceded by space.

2.2.8.10 v1.1.0 (2021-02-16)**Feat**

- (look) #57: Add *SINGLE* function.
- (function) #51: Add google Excel functions.
- (logic) #55, #57: Add IFS function.
- (excel) #65: Add documentation and rename method to load models from ranges.
- (excel) #65: Add method to load sub-models from range.
- (doc): Update Copyright.
- (excel): Improve performances.
- (excel) #64: Read model from outputs.
- (core): Update range definition with path file.
- (excel) #64: Add warning for missing reference.
- (excel) #64: Add warning message when book loading fails.
- (readme) #44: Add example to export and import the model to JSON format.
- (readme) #53: Add instructions to install the development version.
- (excel) #44: Add feature to export and import the model to JSON- able dict.
- (stat, comp) #43: Add *STDEV*, *STDEV.S*, *STDEV.P*, *STDEVA*, *STDEVPA*, *VAR*, *VAR.S*, *VAR.P*, *VARA*, and *VARPA* functions.

Fix

- (financial): Correct requirements for *irr* function.
- (excel) #48: Correct reference pointing to different workbooks.
- (function) #67: Correct compilation of impure functions (e.g., *rand*, *now*, etc.).
- (look) #66: Correct *check* function did not return value.
- (test): Remove *temp* dir.
- (excel): Correct external link reading.
- (operator) #63: Correct operator parser when starts with spaces.
- (text) #61: Convert float as int when stringify if it is an integer.
- (math) #59: Convert string to number in math operations.
- (functions): Correct *_xfilter* operating range type.

- (parser) #61: Skip n in formula expression.
- (operator) #58: Correct operator parser for composed operators.
- (excel): Correct invalid range definition and missing sheet or files.
- (operand) #52: Correct range parser.
- (operand) #50: Correct sheet name parser with space.
- (tokens): Correct closure parenthesis parser.
- (excel): Skip function compilation for string cells.
- (tokens): Correct error parsing when sheet name is defined.

2.2.8.11 v1.0.0 (2020-03-12)

Feat

- (core): Add *CODE_OF_CONDUCT.md*.
- (function) #39: Transform *NotImplementedError* into *#NAME?*.
- (text) #39: Add *CONCAT* and *CONCATENATE* functions.
- (logic) #38: Add *TRUE/FALSE* functions.
- (excel) #42: Save missing nodes.
- (excel) #42: Update logic for *RangesAssembler*.
- (excel): Improve performance of *finish* method.
- (core): Update build script.
- (core): Add support for python 3.8 and drop python 3.5 and drop *appveyor*.
- (core): Improve memory performance.
- (refact): Update copyright.
- (operand): Add *fast_range2parts_v4* for named ranges.

Fix

- (math) #37: Match excel default rounding algorithm of round half up.
- (cell): Correct reference in *push* method.
- (readme): Correct doctest.
- (token): Correct separator parser.
- (excel) #35: Update logic to parse named ranges.
- (operand): Associate *excel_id==0* to current excel.
- (array): Ensure correct deepcopy of *Array* attributes.
- (operand) #39: Correct range parser for named ranges.
- (operand) #41: Correct named ranges parser.

2.2.8.12 v0.4.0 (2019-08-31)

Feat

- (doc): Add binder.
- (setup): Add env *ENABLE_SETUP_LONG_DESCRIPTION*.
- (core): Add useful constants.
- (excel): Add option to write all calculate books inside a folder.
- (stat) #21: Add *COUNTBLANK*, *LARGE*, *SMALL* functions.
- (date) #35: Add *NPV*, *XNPV*, *IRR*, *XIRR* functions.
- (stat) #21: Add *AVERAGEIF*, *COUNT*, *COUNTA*, *COUNTIF* functions.
- (math) #21: Add *SUMIF* function.
- (date) #21, #35, #36: Add *date* functions *DATE*, *DATEVALUE*, *DAY*, *MONTH*, *YEAR*, *TODAY*, *TIME*, *TIMEVALUE*, *SECOND*, *MINUTE*, *hour*, *NOW*, *YEARFRAC*.
- (info) #21: Add *NA* function.
- (date) #21, #35, #36: Add *date* functions *DATE*, *DATEVALUE*, *DAY*, *MONTH*, *YEAR*, *TODAY*, *TIME*, *TIMEVALUE*, *SECOND*, *MINUTE*, *hour*, *NOW*, *YEARFRAC*.
- (stat) #35: Add *MINA*, *AVERAGEA*, *MAXA* functions.

Fix

- (setup): Update tests requirements.
- (setup): Correct setup dependency (*beautifulsoup4*).
- (stat): Correct round indices.
- (setup) #34: Build universal wheels.
- (test): Correct import error.
- (date) #35: Correct behaviour of *LOOKUP* function when dealing with errors.
- (excel) #35: Improve cycle detection.
- (excel,date) #21, #35: Add custom Excel Reader to parse raw datetime.
- (excel) #35: Correct when definedName is relative *#REF!*.

2.2.8.13 v0.3.0 (2019-04-24)

Feat

- (logic) #27: Add *OR*, *XOR*, *AND*, *NOT* functions.
- (look) #27: Add *INDEX* function.
- (look) #24: Improve performances of *look* functions.
- (functions) #26: Add *SWITCH*.
- (functions) #30: Add *GCD* and *LCM*.

- (chore): Improve performances avoiding *combine_dicts*.
- (chore): Improve performances checking intersection.

Fix

- (tokens): Correct string nodes ids format adding “.
- (ranges): Correct behaviour union of ranges.
- (import): Enable PyCharm autocomplete.
- (import): Save imports.
- (test): Add repo path to system path.
- (parser): Parse empty args for functions.
- (functions) #30: Correct implementation of *GCD* and *LCM*.
- (ranges) #24: Enable full column and row reference.
- (excel): Correct bugs due to new *openpyxl*.

2.2.8.14 v0.2.0 (2018-12-11)

Feat

- (doc) #23: Enhance *ExcelModel* documentation.

Fix

- (core): Add python 3.7 and drop python 3.4.
- (excel): Make *ExcelModel* dillable and pickable.
- (builder): Avoid *FormulaError* exception during formulas compilation.
- (excel): Correct bug when compiling excel with circular references.

2.2.8.15 v0.1.4 (2018-10-19)

Fix

- (tokens) #20: Improve Number regex.

2.2.8.16 v0.1.3 (2018-10-09)

Feat

- (excel) #16: Solve circular references.
- (setup): Add donate url.

Fix

- (functions) #18: Enable *check_error* in *IF* function just for the first argument.
- (functions) #18: Disable *input_parser* in *IF* function to return any type of values.
- (rtd): Define *fpath* from *prj_dir* for rtd.
- (rtd): Add missing requirements *openpyxl* for rtd.
- (setup): Patch to use *sphinxcontrib.restbuilder* in setup *long_description*.

Other

- Update documentation.
- Replace *excel* with *Excel*.
- Create `PULL_REQUEST_TEMPLATE.md`.
- Update issue templates.
- Update copyright.
- (doc): Update author mail.

2.2.8.17 v0.1.2 (2018-09-12)

Feat

- (functions) #14: Add *ROW* and *COLUMN*.
- (cell): Pass cell reference when compiling cell + new function struct with dict to add inputs like *CELL*.

Fix

- (ranges): Replace system max size with excel max row and col.
- (tokens): Correct number regex.

2.2.8.18 v0.1.1 (2018-09-11)

Feat

- (contrib): Add contribution instructions.
- (setup): Add additional project_urls.
- (setup): Update *Development Status* to 4 - *Beta*.

Fix

- (init) #15: Replace *FUNCTIONS* and *OPERATORS* objs with *get_functions*, *SUBMODULES*.
- (doc): Correct link docs_status.

2.2.8.19 v0.1.0 (2018-07-20)

Feat

- (readme) #6, #7: Add examples.
- (doc): Add changelog.
- (test): Add info of executed test of *test_excel_model*.
- (functions) #11: Add *HEX2OCT*, *HEX2BIN*, *HEX2DEC*, *OCT2HEX*, *OCT2BIN*, *OCT2DEC*, *BIN2HEX*, *BIN2OCT*, *BIN2DEC*, *DEC2HEX*, *DEC2OCT*, and *DEC2BIN* functions.
- (setup) #13: Add *extras_require* to setup file.

Fix

- (excel): Use *DispatchPipe* to compile a sub model of excel workbook.
- (range) #11: Correct range regex to avoid parsing of function like ranges (e.g., *HEX2DEC*).

2.2.8.20 v0.0.10 (2018-06-05)

Feat

- (look): Simplify *_get_type_id* function.

Fix

- (functions): Correct *ImportError* for *FUNCTIONS*.
- (operations): Correct behaviour of the basic operations.

2.2.8.21 v0.0.9 (2018-05-28)

Feat

- (excel): Improve performances pre-calculating the range format.
- (core): Improve performances using *DispatchPipe* instead *SubDispatchPipe* when compiling formulas.
- (function): Improve performances setting *errstate* outside vectorization.
- (core): Improve performances of *range2parts* function (overall 50% faster).

Fix

- (ranges): Minimize conversion str to int and vice versa.
- (functions) #10: Avoid returning shapeless array.

2.2.8.22 v0.0.8 (2018-05-23)

Feat

- (functions): Add *MATCH*, *LOOKUP*, *HLOOKUP*, *VLOOKUP* functions.
- (excel): Add method to compile *ExcelModel*.
- (travis): Run coveralls in python 3.6.
- (functions): Add *FIND*, *LEFT*, *LEN*, *LOWER*, *MID*, *REPLACE*, *RIGHT*, *TRIM*, and *UPPER* functions.
- (functions): Add *IRR* function.
- (formulas): Custom reshape to Array class.
- (functions): Add *ISO.CEILING*, *SQRTPI*, *TRUNC* functions.
- (functions): Add *ROUND*, *ROUNDDOWN*, *ROUNDUP*, *SEC*, *SECH*, *SIGN* functions.
- (functions): Add *DECIMAL*, *EVEN*, *MROUND*, *ODD*, *RAND*, *RANDBETWEEN* functions.
- (functions): Add *FACT* and *FACTDOUBLE* functions.
- (functions): Add *ARABIC* and *ROMAN* functions.
- (functions): Parametrize function *wrap_ufunc*.
- (functions): Split function *raise_errors* adding *get_error* function.
- (ranges): Add custom default and error value for defining ranges Arrays.
- (functions): Add *LOG10* function + fix *LOG*.
- (functions): Add *CSC* and *CSCH* functions.
- (functions): Add *COT* and *COTH* functions.
- (functions): Add *FLOOR*, *FLOOR.MATH*, and *FLOOR.PRECISE* functions.
- (test): Improve log message of test cell.

Fix

- (rtd): Update installation file for read the docs.
- (functions): Remove unused functions.
- (formulas): Avoid too broad exception.
- (functions.math): Drop scipy dependency for calculate factorial2.
- (functions.logic): Correct error behaviour of *if* and *iferror* functions + add BroadcastError.
- (functions.info): Correct behaviour of *iserr* function.
- (functions): Correct error behaviour of average function.
- (functions): Correct *iserror* and *iserr* returning a custom Array.
- (functions): Now *xceiling* function returns np.nan instead Error.errors['#NUM!'].
- (functions): Correct *is_number* function, now returns False when number is a bool.
- (test): Ensure same order of workbook comparisons.
- (functions): Correct behaviour of *min max* and *int* function.
- (ranges): Ensure to have a value with correct shape.
- (parser): Change order of parsing to avoid TRUE and FALSE parsed as ranges or errors as strings.
- (function): Remove unused kwargs n_out.
- (parser): Parse error string as formulas.
- (readme): Remove *downloads_count* because it is no longer available.

Other

- Refact: Update Copyright + minor pep.
- Excel returns 1-indexed string positions???
- Added common string functions.
- Merge pull request #9 from ecatkins/irr.
- Implemented IRR function using numpy.

2.2.8.23 v0.0.7 (2017-07-20)

Feat

- (appveyor): Add python 3.6.
- (functions) #4: Add *sumproduct* function.

Fix

- (install): Force update setuptools>=36.0.1.
- (functions): Correct *iserror* *iserr* functions.
- (ranges): Replace '#N/A' with '' as empty value when assemble values.
- (functions) #4: Remove check in ufunc when inputs have different size.
- (functions) #4: Correct *power*, *arctan2*, and *mod* error results.
- (functions) #4: Simplify ufunc code.
- (test) #4: Check that all results are in the output.
- (functions) #4: Correct *atan2* argument order.
- (range) #5: Avoid parsing function name as range when it is followed by (.
- (operator) #3: Replace *strip* with *replace*.
- (operator) #3: Correct valid operators like ^- or *+.

Other

- Made the ufunc wrapper work with multi input functions, e.g., *power*, *mod*, and *atan2*.
- Created a workbook comparison method in *TestExcelModel*.
- Added MIN and MAX to the test.xlsx.
- Cleaned up the ufunc wrapper and added min and max to the functions list.
- Relaxed equality in *TestExcelModel* and made some small fixes to *functions.py*.
- Added a wrapper for numpy ufuncs, mapped some Excel functions to ufuncs and provided tests.

2.2.8.24 v0.0.6 (2017-05-31)

Fix

- (plot): Update schedula to 0.1.12.
- (range): Sheet name without commas has this [^Wd][w.] format.

2.2.8.25 v0.0.5 (2017-05-04)

Fix

- (doc): Update schedula to 0.1.11.

2.2.8.26 v0.0.4 (2017-02-10)

Fix

- (regex): Remove deprecation warnings.

2.2.8.27 v0.0.3 (2017-02-09)

Fix

- (appveyor): Setup of lxml.
- (excel): Remove deprecation warning openpyxl.
- (requirements): Update schedula requirement 0.1.9.

2.2.8.28 v0.0.2 (2017-02-08)

Fix

- (setup): setup fails due to long description.
- (excel): Remove deprecation warning *remove_sheet* → *remove*.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

f

- `formulas`, 15
- `formulas.builder`, 17
- `formulas.cell`, 303
- `formulas.errors`, 19
- `formulas.excel`, 308
- `formulas.excel.cycle`, 308
- `formulas.excel.xlreader`, 308
- `formulas.functions`, 60
- `formulas.functions.comp`, 60
- `formulas.functions.date`, 60
- `formulas.functions.eng`, 63
- `formulas.functions.financial`, 63
- `formulas.functions.google`, 64
- `formulas.functions.info`, 65
- `formulas.functions.logic`, 242
- `formulas.functions.look`, 243
- `formulas.functions.math`, 245
- `formulas.functions.operators`, 249
- `formulas.functions.stat`, 249
- `formulas.functions.text`, 251
- `formulas.parser`, 16
- `formulas.ranges`, 301
- `formulas.tokens`, 20
- `formulas.tokens.function`, 21
- `formulas.tokens.operand`, 24
- `formulas.tokens.operator`, 47
- `formulas.tokens.parenthesis`, 56

Symbols

__init__() (Array method), 22, 289
 __init__() (AstBuilder method), 18
 __init__() (Cell method), 305
 __init__() (CellWrapper method), 305
 __init__() (Empty method), 27
 __init__() (Error method), 29
 __init__() (ExcelModel method), 315
 __init__() (Function method), 24
 __init__() (Intersect method), 49
 __init__() (IsErrArray method), 99
 __init__() (IsErrorArray method), 143
 __init__() (IsNaArray method), 187
 __init__() (IsNumberArray method), 231
 __init__() (Number method), 31
 __init__() (Operand method), 33
 __init__() (Operator method), 51
 __init__() (OperatorToken method), 53
 __init__() (Parenthesis method), 57
 __init__() (Parser method), 16
 __init__() (Range method), 34
 __init__() (Ranges method), 302
 __init__() (RangesAssembler method), 306
 __init__() (Ref method), 307
 __init__() (Separator method), 55
 __init__() (String method), 36
 __init__() (Token method), 59
 __init__() (XlCircular method), 326
 __init__() (XlError method), 47
 __init__() (XlReader method), 310

A

args2list() (in module *formulas.functions*), 254
 args2vals() (in module *formulas.functions*), 254
 Array (class in *formulas.functions*), 256
 Array (class in *formulas.tokens.function*), 21
 AstBuilder (class in *formulas.builder*), 17

C

Cell (class in *formulas.cell*), 304
 CellWrapper (class in *formulas.cell*), 305
 clean_values() (in module *formulas.functions*), 254

compile_class (AstBuilder attribute), 18
 compile_class (ExcelModel attribute), 316
 convert2float() (in module *formulas.functions*), 254
 convert_nan() (in module *formulas.functions*), 254
 convert_noshp() (in module *formulas.functions*), 254

E

Empty (class in *formulas.tokens.operand*), 26
 Error (class in *formulas.tokens.operand*), 28
 ExcelModel (class in *formulas.excel*), 311

F

fast_range2parts() (in module *formulas.tokens.operand*), 25
 fast_range2parts_v1() (in module *formulas.tokens.operand*), 25
 fast_range2parts_v2() (in module *formulas.tokens.operand*), 25
 fast_range2parts_v3() (in module *formulas.tokens.operand*), 25
 fast_range2parts_v4() (in module *formulas.tokens.operand*), 25
 fast_range2parts_v5() (in module *formulas.tokens.operand*), 26
 flatten() (in module *formulas.functions*), 254
 format_output() (in module *formulas.cell*), 303
 formulas
 module, 15
 formulas.builder
 module, 17
 formulas.cell
 module, 303
 formulas.errors
 module, 19
 formulas.excel
 module, 308
 formulas.excel.cycle
 module, 308
 formulas.excel.xlreader
 module, 308
 formulas.functions
 module, 60

[formulas.functions.comp](#)
 module, 60
[formulas.functions.date](#)
 module, 60
[formulas.functions.eng](#)
 module, 63
[formulas.functions.financial](#)
 module, 63
[formulas.functions.google](#)
 module, 64
[formulas.functions.info](#)
 module, 65
[formulas.functions.logic](#)
 module, 242
[formulas.functions.look](#)
 module, 243
[formulas.functions.math](#)
 module, 245
[formulas.functions.operators](#)
 module, 249
[formulas.functions.stat](#)
 module, 249
[formulas.functions.text](#)
 module, 251
[formulas.parser](#)
 module, 16
[formulas.ranges](#)
 module, 301
[formulas.tokens](#)
 module, 20
[formulas.tokens.function](#)
 module, 21
[formulas.tokens.operand](#)
 module, 24
[formulas.tokens.operator](#)
 module, 47
[formulas.tokens.parenthesis](#)
 module, 56
[Function](#) (class in [formulas.tokens.function](#)), 23

G

[get_error\(\)](#) (in module [formulas.functions](#)), 254
[get_functions\(\)](#) (in module [formulas.functions](#)), 254

H

[hex2dec2bin2oct\(\)](#) (in module [formulas.functions.eng](#)), 63

I

[Intersect](#) (class in [formulas.tokens.operator](#)), 47
[is_not_empty\(\)](#) (in module [formulas.functions](#)), 255
[is_number\(\)](#) (in module [formulas.functions](#)), 255
[iserr\(\)](#) (in module [formulas.functions.info](#)), 65

[IsErrArray](#) (class in [formulas.functions.info](#)), 66
[iserror\(\)](#) (in module [formulas.functions.info](#)), 65
[IsErrorArray](#) (class in [formulas.functions.info](#)), 110
[isna\(\)](#) (in module [formulas.functions.info](#)), 65
[IsNaArray](#) (class in [formulas.functions.info](#)), 154
[IsNumberArray](#) (class in [formulas.functions.info](#)), 198

L

[load_workbook\(\)](#) (in module [formulas.excel.xlreader](#)), 309
[logic_input_parser\(\)](#) (in module [formulas.functions.operators](#)), 249

M

[module](#)
 [formulas](#), 15
 [formulas.builder](#), 17
 [formulas.cell](#), 303
 [formulas.errors](#), 19
 [formulas.excel](#), 308
 [formulas.excel.cycle](#), 308
 [formulas.excel.xlreader](#), 308
 [formulas.functions](#), 60
 [formulas.functions.comp](#), 60
 [formulas.functions.date](#), 60
 [formulas.functions.eng](#), 63
 [formulas.functions.financial](#), 63
 [formulas.functions.google](#), 64
 [formulas.functions.info](#), 65
 [formulas.functions.logic](#), 242
 [formulas.functions.look](#), 243
 [formulas.functions.math](#), 245
 [formulas.functions.operators](#), 249
 [formulas.functions.stat](#), 249
 [formulas.functions.text](#), 251
 [formulas.parser](#), 16
 [formulas.ranges](#), 301
 [formulas.tokens](#), 20
 [formulas.tokens.function](#), 21
 [formulas.tokens.operand](#), 24
 [formulas.tokens.operator](#), 47
 [formulas.tokens.parenthesis](#), 56

N

[not_implemented\(\)](#) (in module [formulas.functions](#)), 255
[Number](#) (class in [formulas.tokens.operand](#)), 30

O

[Operand](#) (class in [formulas.tokens.operand](#)), 32
[Operator](#) (class in [formulas.tokens.operator](#)), 50
[OperatorToken](#) (class in [formulas.tokens.operator](#)), 52

P

Parenthesis (class in *formulas.tokens.parenthesis*), 56
 parse_ranges() (in module *formulas.functions*), 255
 Parser (class in *formulas.parser*), 16

R

raise_errors() (in module *formulas.functions*), 255
 Range (class in *formulas.tokens.operand*), 33
 range2parts() (in module *formulas.tokens.operand*), 26
 Ranges (class in *formulas.ranges*), 301
 RangesAssembler (class in *formulas.cell*), 306
 Ref (class in *formulas.cell*), 307
 replace_empty() (in module *formulas.functions*), 255
 reshape() (Array method), 300
 round_up() (in module *formulas.functions.math*), 247

S

Separator (class in *formulas.tokens.operator*), 54
 simple_cycles() (in module *formulas.excel.cycle*), 308
 solve_cycle() (in module *formulas.functions.logic*), 243
 String (class in *formulas.tokens.operand*), 35

T

text2num() (in module *formulas.functions*), 255
 to_number() (in module *formulas.functions*), 255
 Token (class in *formulas.tokens*), 58

V

value_return() (in module *formulas.functions*), 255

W

wrap_cell_func() (in module *formulas.cell*), 303
 wrap_func() (in module *formulas.functions*), 256
 wrap_impure_func() (in module *formulas.functions*), 256
 wrap_ranges_func() (in module *formulas.functions*), 256
 wrap_ufunc() (in module *formulas.functions*), 256

X

xaddress() (in module *formulas.functions.look*), 244
 xand() (in module *formulas.functions.logic*), 243
 xarabic() (in module *formulas.functions.math*), 247
 xarctan2() (in module *formulas.functions.math*), 247
 xceiling() (in module *formulas.functions.math*), 247
 xceiling_math() (in module *formulas.functions.math*), 247
 xcolumn() (in module *formulas.functions.look*), 244
 xconcat() (in module *formulas.functions.text*), 251
 xcorrel() (in module *formulas.functions.stat*), 250
 xcot() (in module *formulas.functions.math*), 247

xcumipmt() (in module *formulas.functions.financial*), 64
 xdate() (in module *formulas.functions.date*), 61
 xdatedif() (in module *formulas.functions.date*), 61
 xdatevalue() (in module *formulas.functions.date*), 61
 xday() (in module *formulas.functions.date*), 62
 xdecimal() (in module *formulas.functions.math*), 247
 xdummy() (in module *formulas.functions.google*), 65
 xedate() (in module *formulas.functions.date*), 62
 xeven() (in module *formulas.functions.math*), 247
 xfact() (in module *formulas.functions.math*), 247
 xfactdouble() (in module *formulas.functions.math*), 248
 xfilter() (in module *formulas.functions*), 256
 xfind() (in module *formulas.functions.text*), 251
 xforecast() (in module *formulas.functions.stat*), 250
 xfunc() (in module *formulas.functions.stat*), 250
 xgcd() (in module *formulas.functions.math*), 248
 xif() (in module *formulas.functions.logic*), 243
 xiferror() (in module *formulas.functions.logic*), 243
 xiferror_return() (in module *formulas.functions.logic*), 243
 xifna() (in module *formulas.functions.logic*), 243
 xifs() (in module *formulas.functions.logic*), 243
 xindex() (in module *formulas.functions.look*), 244
 xirr() (in module *formulas.functions.financial*), 64
 xiseven_odd() (in module *formulas.functions.info*), 66
 xisoweeknum() (in module *formulas.functions.date*), 62
 XlCircular (class in *formulas.excel*), 316
 xlcm() (in module *formulas.functions.math*), 248
 xleft() (in module *formulas.functions.text*), 251
 XLError (class in *formulas.tokens.operand*), 37
 xlookup() (in module *formulas.functions.look*), 244
 XlReader (class in *formulas.excel.xlreader*), 309
 xmatch() (in module *formulas.functions.look*), 244
 xmid() (in module *formulas.functions.text*), 251
 xmod() (in module *formulas.functions.math*), 248
 xmround() (in module *formulas.functions.math*), 248
 xna() (in module *formulas.functions.info*), 66
 xnow() (in module *formulas.functions.date*), 62
 xnper() (in module *formulas.functions.financial*), 64
 xnpv() (in module *formulas.functions.financial*), 64
 xodd() (in module *formulas.functions.math*), 248
 xpower() (in module *formulas.functions.math*), 248
 xppmt() (in module *formulas.functions.financial*), 64
 xrandbetween() (in module *formulas.functions.math*), 248
 xrate() (in module *formulas.functions.financial*), 64
 xreplace() (in module *formulas.functions.text*), 252
 xright() (in module *formulas.functions.text*), 252
 xroman() (in module *formulas.functions.math*), 248
 xround() (in module *formulas.functions.math*), 249
 xrow() (in module *formulas.functions.look*), 245
 xsearch() (in module *formulas.functions.text*), 252
 xsecond() (in module *formulas.functions.date*), 62

`xsingle()` (in module `formulas.functions.look`), 245
`xslope()` (in module `formulas.functions.stat`), 250
`xsort()` (in module `formulas.functions.stat`), 250
`xsqrtpi()` (in module `formulas.functions.math`), 249
`xstdev()` (in module `formulas.functions.stat`), 250
`xsum()` (in module `formulas.functions.math`), 249
`xsumproduct()` (in module `formulas.functions.math`),
249
`xswitch()` (in module `formulas.functions.logic`), 243
`xtext()` (in module `formulas.functions.text`), 252
`xtime()` (in module `formulas.functions.date`), 62
`xtimevalue()` (in module `formulas.functions.date`), 62
`xtoday()` (in module `formulas.functions.date`), 62
`xtrunc()` (in module `formulas.functions.math`), 249
`xvalue()` (in module `formulas.functions.text`), 252
`xweekday()` (in module `formulas.functions.date`), 62
`xweeknum()` (in module `formulas.functions.date`), 63
`xxirr()` (in module `formulas.functions.financial`), 64
`xxnpv()` (in module `formulas.functions.financial`), 64
`xyearfrac()` (in module `formulas.functions.date`), 63