formulas Documentation

Release 0.1.4

Vincenzo Arcidiacono

Oct 19, 2018

Table of Contents

2018-10-19 11:00:00

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>

CHAPTER 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**.

CHAPTER 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
```

2.1 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]
```

2.1.1 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**.

2.1.2 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
```

2.1.2.1 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]
```

2.1.3 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.

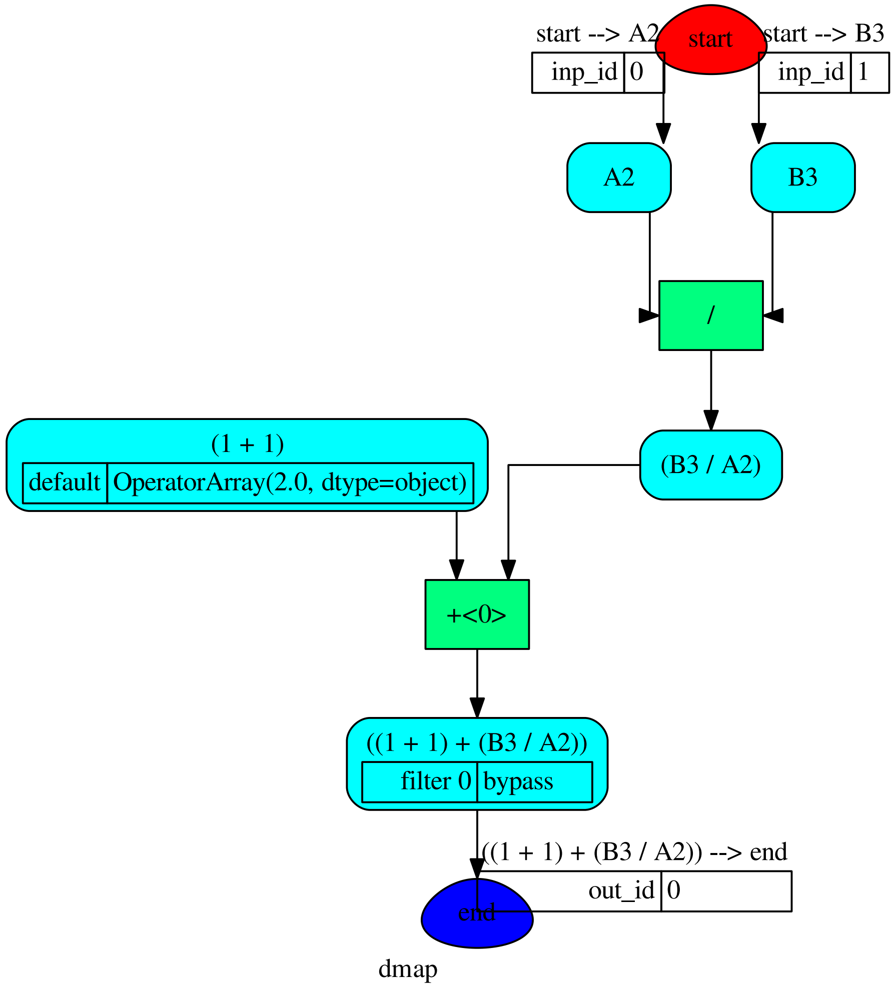
2.1.3.1 Parsing formula

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()
```

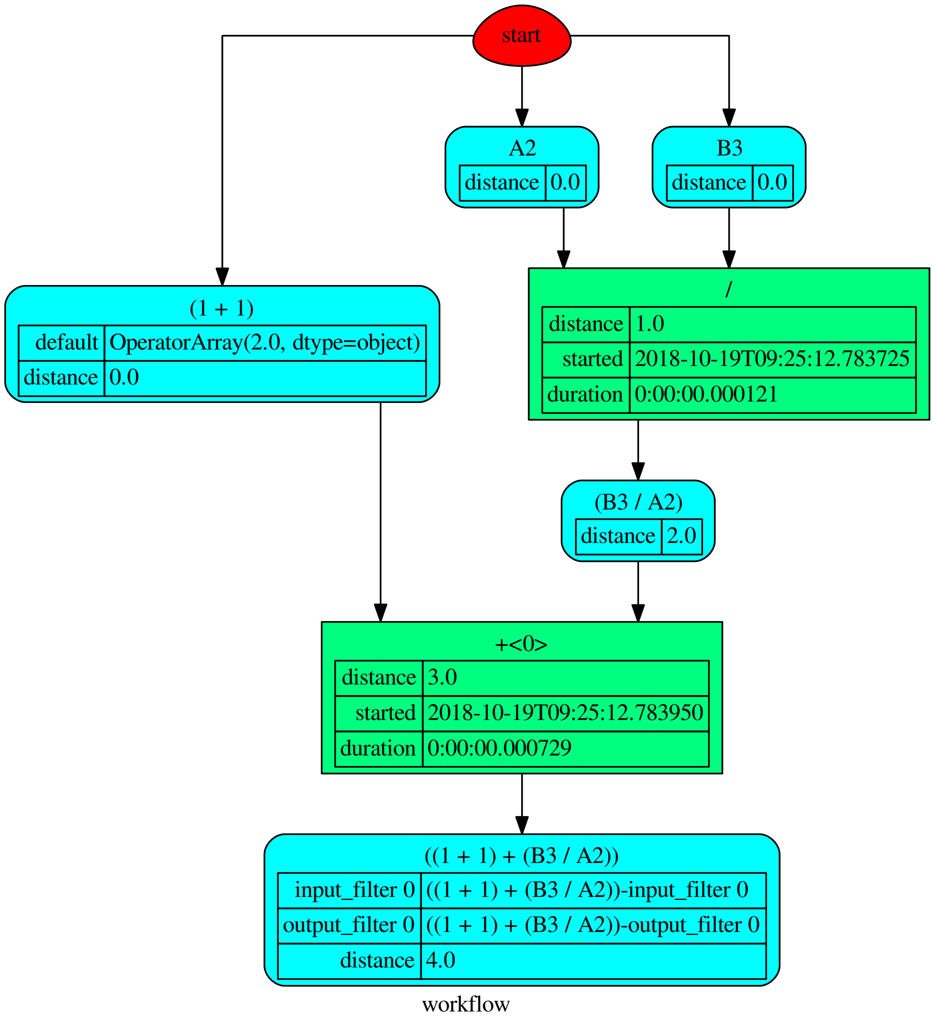
To visualize formula model and get the input order you can do the following:

```
>>> 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)
OperatorArray(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())])
```



2.1.3.2 Excel workbook

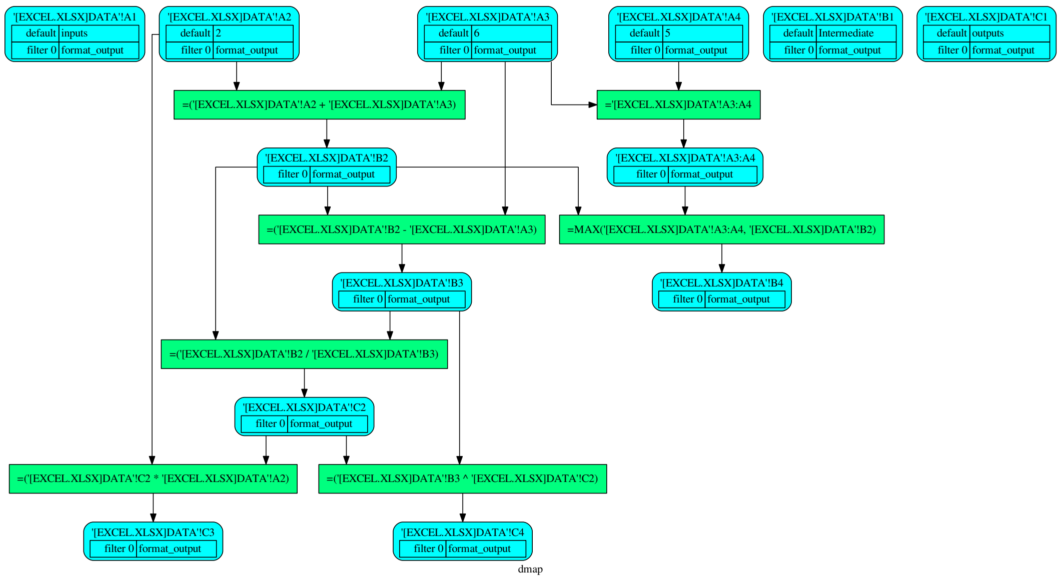
An example how to load, calculate, and write an Excel workbook is the following:

```
>>> import formulas
>>> fpath = 'file.xlsx'
>>> xl_model = formulas.ExcelModel().loads(fpath).finish()
>>> xl_model.calculate()
Solution(...)
>>> xl_model.write()
{'EXCEL.XLSX': {Book: <openpyxl.workbook.workbook.Workbook ...>}}
```

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

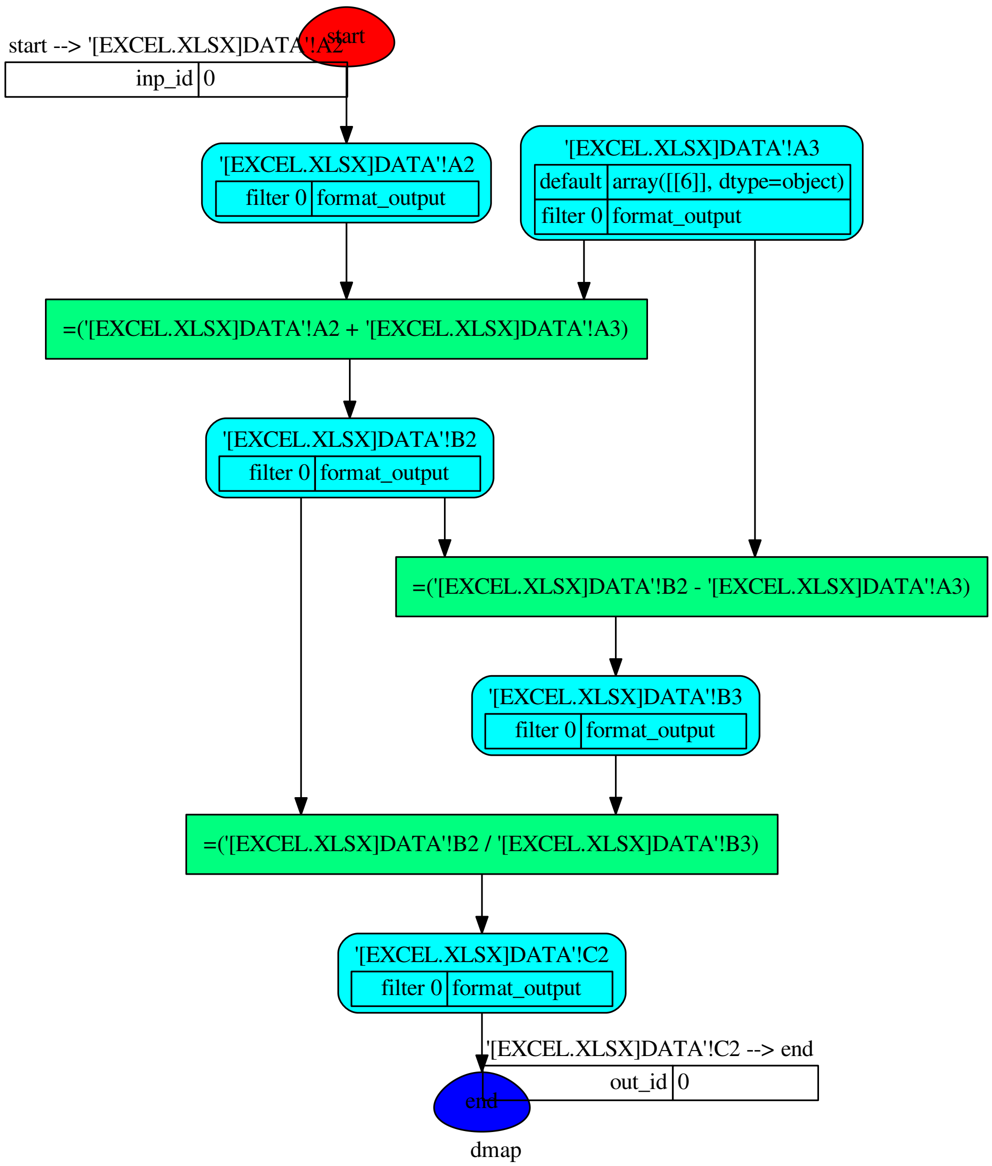
To plot the dependency graph that depict relationships between Excel cells:

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



To compile, execute, and plot a Excel sub-model you can do the following:

```
>>> inputs = ["'[EXCEL.XLSX]DATA'!A2"]  # input cells
>>> outputs = ["'[EXCEL.XLSX]DATA'!C2"]  # output cells
>>> func = xl_model.compile(inputs, outputs)
>>> func(2).value[0,0]
4.0
>>> func.plot(view=False)  # Set view=True to plot in the default browser.
SiteMap([(Dispatcher ..., SiteMap())])
```



2.1.3.3 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
```

2.1.4 Next moves

Things yet to do: implement the missing Excel formulas.

2.1.5 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

2.1.5.1 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`.

2.1.5.2 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.

2.1.5.3 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).

2.1.6 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	Implementation order
•	150€	120€	50€	•	•
SUM	50€	40€	25€	125€	1st
SIN	50€		25€	75€	3rd
TAN	50€	40€		90€	2nd
COS		40€		40€	4th

2.1.7 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:

`parser`	It provides formula parser class.
`builder`	It provides AstBuilder class.
`errors`	Defines the formulas exception.
`tokens`	It provides tokens needed to parse the Excel formulas.
`functions`	It provides functions implementations to compile the Excel functions.

Continued on next page

Table 1 – continued from previous page

ranges	It provides Ranges class.
cell	It provides Cell class.
excel	It provides Excel model class.

2.1.7.1 parser

It provides formula parser class.

Classes

Parser

Parser

class Parser

Methods

ast
is_formula

ast

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

is_formula

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

__init__()

Initialize self. See help(type(self)) for accurate signature.

Attributes

filters
formula_check

filters

Parser.**filters = [<class 'formulas.tokens.operand.Error'>, <class 'formulas.tokens.ope**

formula_check

Parser.**formula_check = regex.Regex('\n (?P<array>^\\s*{\\s*=\\s*(?P<name>\\S.*)\\s*}**

ast_builder
alias of *formulas.builder.AstBuilder*

2.1.7.2 builder

It provides AstBuilder class.

Classes

AstBuilder

AstBuilder

class AstBuilder(**args*, *dsp=None*, *nodes=None*, *match=None*, ***kwargs*)

Methods

__init__	Initialize self.
append	Add an element to the right side of the deque.
appendleft	Add an element to the left side of the deque.
clear	Remove all elements from the deque.
compile	
copy	Return a shallow copy of a deque.
count	
extend	Extend the right side of the deque with elements from the iterable
extendleft	Extend the left side of the deque with elements from the iterable
finish	
get_node_id	
index	Raises ValueError if the value is not present.
insert	D.insert(index, object) – insert object before index
pop	Remove and return the rightmost element.
popleft	Remove and return the leftmost element.
remove	D.remove(value) – remove first occurrence of value.
reverse	D.reverse() – reverse *IN PLACE*
rotate	Rotate the deque n steps to the right (default n=1).

__init__

AstBuilder.**__init__**(**args*, *dsp=None*, *nodes=None*, *match=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

append

AstBuilder.**append**(*token*)
Add an element to the right side of the deque.

appendleft

AstBuilder.**appendleft**()
Add an element to the left side of the deque.

clear

AstBuilder.**clear**()
Remove all elements from the deque.

compile

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

copy

AstBuilder.**copy**()
Return a shallow copy of a deque.

count

AstBuilder.**count**(*value*) → integer – return number of occurrences of value

extend

AstBuilder.**extend**()
Extend the right side of the deque with elements from the iterable

extendleft

AstBuilder.**extendleft**()
Extend the left side of the deque with elements from the iterable

finish

AstBuilder.**finish**()

get_node_id

AstBuilder.**get_node_id**(*token*)

index

AstBuilder.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

insert

`AstBuilder.`**`insert`**`()`
D.insert(index, object) – insert object before index

pop

`AstBuilder.`**`pop`**`()`
Remove and return the rightmost element.

popleft

`AstBuilder.`**`popleft`**`()`
Remove and return the leftmost element.

remove

`AstBuilder.`**`remove`**`()`
D.remove(value) – remove first occurrence of value.

reverse

`AstBuilder.`**`reverse`**`()`
D.reverse() – reverse *IN PLACE*

rotate

`AstBuilder.`**`rotate`**`()`
Rotate the deque n steps to the right (default n=1). If n is negative, rotates left.

`__init__``(`**args*, *dsp=None*, *nodes=None*, *match=None*, ***kwargs*`)`
Initialize self. See help(type(self)) for accurate signature.

Attributes

`maxlen`	maximum size of a deque or None if unbounded

maxlen

`AstBuilder.`**`maxlen`**
maximum size of a deque or None if unbounded

`append``(`*token*`)`
Add an element to the right side of the deque.

2.1.7.3 errors

Defines the formulas exception.

Exceptions

BaseError
BroadcastError
FormulaError
FoundError
FunctionError
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*)

ParenthesesError

exception ParenthesesError(**args*)

RangeValueError

exception RangeValueError(**args*)

TokenError

exception TokenError(**args*)

2.1.7.4 tokens

It provides tokens needed to parse the Excel formulas.

Sub-Modules:

function	It provides Function classes.
operand	It provides Operand classes.
operator	It provides Operator classes.
parenthesis	It provides Parenthesis class.

function

It provides Function classes.

Classes

Array
Function

Array

class Array(*s, context=None*)

Methods

__init__	Initialize self.
ast	
compile	
match	
process	
set_expr	
update_input_tokens	

__init__

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

Initialize self. See help(type(self)) for accurate signature.

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*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

name	
node_id	

name

`Array.name`

node_id

`Array.node_id`

Function

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

Methods

__init__	Initialize self.
ast	

Continued on next page

Table 13 – continued from previous page

compile
match
process
set_expr
update_input_tokens

__init__

`Function.__init__(s, context=None)`
Initialize self. See help(type(self)) for accurate signature.

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)`
Initialize self. See help(type(self)) for accurate signature.

Attributes

name
node_id

name

`Function.name`

node_id

`Function.node_id`

operand

It provides Operand classes.

Functions

fast_range2parts
fast_range2parts_v1
fast_range2parts_v2
fast_range2parts_v3
range2parts

fast_range2parts

fast_range2parts(***kw*)

fast_range2parts_v1

fast_range2parts_v1(*r1*, *c1*, *excel*, *sheet=''*)

fast_range2parts_v2

fast_range2parts_v2(*r1*, *c1*, *r2*, *c2*, *excel*, *sheet=''*)

fast_range2parts_v3

fast_range2parts_v3(*r1*, *n1*, *r2*, *n2*, *excel*, *sheet=''*)

range2parts

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

Classes

Error

Continued on next page

Table 16 – continued from previous page

Number
Operand
Range
String
XlError

Error

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

Methods

__init__	Initialize self.
ast	
compile	
match	
process	
set_expr	
update_input_tokens	

__init__

Error.**__init__**(*s*, *context=None*)
Initialize self. See help(type(self)) for accurate signature.

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*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

errors
k
name
node_id

errors

Error.**errors = {'#DIV/0!': #DIV/0!, '#N/A': #N/A, '#NAME?': #NAME?, '#NULL!': #NULL!,**

k

Error.**k = '#N/A'**

name

Error.**name**

node_id

Error.**node_id**

Number

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

Methods

__init__	Initialize self.
ast	
compile	
match	
process	
set_expr	
update_input_tokens	

__init__

Number.**__init__**(*s*, *context=None*)
Initialize self. See help(type(self)) for accurate signature.

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*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

name
node_id

name

Number.**name**

node_id

Number.**node_id**

Operand

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

Methods

__init__	Initialize self.
ast	
match	
process	
set_expr	
update_input_tokens	

__init__

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

Initialize self. See help(type(self)) for accurate signature.

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*)

Initialize self. See help(type(self)) for accurate signature.

Attributes

name
node_id

name

`Operand.name`

node_id

`Operand.node_id`

Range

`class Range(s, context=None)`

Methods

`__init__`	Initialize self.
`ast`	
`compile`	
`match`	
`process`	
`set_expr`	
`update_input_tokens`	

__init__

`Range.__init__(s, context=None)`
Initialize self. See help(type(self)) for accurate signature.

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*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

name	
node_id	

name

Range.**name**

node_id

Range.**node_id**

String

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

Methods

__init__	Initialize self.
ast	
compile	
match	
process	
set_expr	
update_input_tokens	

__init__

String.**__init__**(*s*, *context=None*)
Initialize self. See help(type(self)) for accurate signature.

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)`
Initialize self. See help(type(self)) for accurate signature.

Attributes

name	
node_id	

name

`String.name`

node_id

`String.node_id`

XlError

`class XlError`

Methods

capitalize	Return a capitalized version of S, i.e.

Continued on next page

Table 27 – continued from previous page

casefold	Return a version of S suitable for caseless comparisons.
center	Return S centered in a string of length width.
count	Return the number of non-overlapping occurrences of substring sub in string S[start:end].
encode	Encode S using the codec registered for encoding.
endswith	Return True if S ends with the specified suffix, False otherwise.
expandtabs	Return a copy of S where all tab characters are expanded using spaces.
find	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
format	Return a formatted version of S, using substitutions from args and kwargs.
format_map	Return a formatted version of S, using substitutions from mapping.
index	Like S.find() but raise ValueError when the substring is not found.
isalnum	Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.
isalpha	Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.
isdecimal	Return True if there are only decimal characters in S, False otherwise.
isdigit	Return True if all characters in S are digits and there is at least one character in S, False otherwise.
isidentifier	Return True if S is a valid identifier according to the language definition.
islower	Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.
isnumeric	Return True if there are only numeric characters in S, False otherwise.
isprintable	Return True if all characters in S are considered printable in repr() or S is empty, False otherwise.
isspace	Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.
istitle	Return True if S is a titlecased string and there is at least one character in S, i.e.
isupper	Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.
join	Return a string which is the concatenation of the strings in the iterable.
ljust	Return S left-justified in a Unicode string of length width.
lower	Return a copy of the string S converted to lowercase.
lstrip	Return a copy of the string S with leading whitespace removed.
maketrans	Return a translation table usable for str.translate().

Continued on next page

Table 27 – continued from previous page

`partition`	Search for the separator sep in S, and return the part before it, the separator itself, and the part after it.
`replace`	Return a copy of S with all occurrences of substring old replaced by new.
`rfind`	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
`rindex`	Like S.rfind() but raise ValueError when the substring is not found.
`rjust`	Return S right-justified in a string of length width.
`rpartition`	Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it.
`rsplit`	Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front.
`rstrip`	Return a copy of the string S with trailing whitespace removed.
`split`	Return a list of the words in S, using sep as the delimiter string.
`splitlines`	Return a list of the lines in S, breaking at line boundaries.
`startswith`	Return True if S starts with the specified prefix, False otherwise.
`strip`	Return a copy of the string S with leading and trailing whitespace removed.
`swapcase`	Return a copy of S with uppercase characters converted to lowercase and vice versa.
`title`	Return a titlecased version of S, i.e.
`translate`	Return a copy of the string S in which each character has been mapped through the given translation table.
`upper`	Return a copy of S converted to uppercase.
`zfill`	Pad a numeric string S with zeros on the left, to fill a field of the specified width.

capitalize

`XlError.`**`capitalize`**`()` → str

Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.

casefold

`XlError.`**`casefold`**`()` → str

Return a version of S suitable for caseless comparisons.

center

`XlError.`**`center`**`(`*width*[, *fillchar*]`)` → str

Return S centered in a 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'*) → bytes

Encode S using the codec registered for encoding. Default encoding is 'utf-8'. errors may be given to set a different error handling scheme. 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*) → str

Return a copy of S 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

Like S.find() but raise ValueError when the substring is not found.

isalnum

`XlError.isalnum()` → bool

Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.

isalpha

`XlError.isalpha()` → bool

Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.

isdecimal

`XlError.isdecimal()` → bool

Return True if there are only decimal characters in S, False otherwise.

isdigit

`XlError.isdigit()` → bool

Return True if all characters in S are digits and there is at least one character in S, False otherwise.

isidentifier

`XlError.isidentifier()` → bool

Return True if S is a valid identifier according to the language definition.

Use keyword.iskeyword() to test for reserved identifiers such as "def" and "class".

islower

`XlError.islower()` → bool

Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

isnumeric

`XlError.isnumeric()` → bool

Return True if there are only numeric characters in S, False otherwise.

isprintable

`XlError.isprintable()` → bool
Return True if all characters in S are considered printable in repr() or S is empty, False otherwise.

isspace

`XlError.isspace()` → bool
Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.

istitle

`XlError.istitle()` → bool
Return True if S is a titlecased string and there is at least one character in S, i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

isupper

`XlError.isupper()` → bool
Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

join

`XlError.join(`*iterable*`)` → str
Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

ljust

`XlError.ljust(`*width*[, *fillchar*]`)` → str
Return S left-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space).

lower

`XlError.lower()` → str
Return a copy of the string S converted to lowercase.

lstrip

`XlError.lstrip(`[*chars*]`)` → str
Return a copy of the string S 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) -> (head*, *sep*, *tail*)

Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.

replace

XlError.**replace**(*old*, *new*[, *count*]) → str

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

Like S.rfind() but raise ValueError when the substring is not found.

rjust

XlError.**rjust**(*width*[, *fillchar*]) → str

Return S right-justified in a string of length width. Padding is done using the specified fill character (default is a space).

rpartition

XlError.**rpartition**(*sep) -> (head*, *sep*, *tail*)

Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

rsplit

`XlError.`**`rsplit`**(*sep=None*, *maxsplit=-1*) → list of strings

Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified, any whitespace string is a separator.

rstrip

`XlError.`**`rstrip`**([*chars*]) → str

Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead.

split

`XlError.`**`split`**(*sep=None*, *maxsplit=-1*) → list of strings

Return a list of the words in S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

splitlines

`XlError.`**`splitlines`**([*keepends*]) → list of strings

Return a list of the lines in S, 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*]) → str

Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead.

swapcase

`XlError.`**`swapcase`**() → str

Return a copy of S with uppercase characters converted to lowercase and vice versa.

title

`XlError.title()` → str

Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

translate

`XlError.translate(table)` → str

Return a copy of the string S in which each character has been mapped through the given translation table. The table must implement lookup/indexing via __getitem__, for instance a dictionary or list, mapping Unicode ordinals to Unicode ordinals, strings, or None. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

upper

`XlError.upper()` → str

Return a copy of S converted to uppercase.

zfill

`XlError.zfill(width)` → str

Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

operator

It provides Operator classes.

Classes

Intersect
Operator
OperatorToken
Separator

Intersect

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

Methods

__init__	Initialize self.
ast	
compile	
match	
process	
set_expr	
update_input_tokens	
update_name	

__init__

Intersect.**__init__**(*s*, *context=None*)
Initialize self. See help(type(self)) for accurate signature.

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*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

get_n_args
name
node_id
pred

get_n_args

Intersect.**get_n_args**

name

Intersect.**name**

node_id

Intersect.**node_id**

pred

Intersect.**pred**

Operator

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

Methods

__init__	Initialize self.
ast	
compile	
match	
process	
set_expr	
update_input_tokens	
update_name	

__init__

Operator.**__init__**(*s*, *context=None*)
Initialize self. See help(type(self)) for accurate signature.

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*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

get_n_args
name
node_id
pred

get_n_args

`Operator.get_n_args`

name

`Operator.name`

node_id

`Operator.node_id`

pred

`Operator.pred`

OperatorToken

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

Methods

__init__	Initialize self.
ast	
compile	
match	
process	
set_expr	
update_input_tokens	
update_name	

__init__

`OperatorToken.__init__`(*s*, *context=None*)
Initialize self. See help(type(self)) for accurate signature.

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*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

get_n_args
name
node_id
pred

get_n_args

OperatorToken.**get_n_args**

name

OperatorToken.**name**

node_id

OperatorToken.**node_id**

pred

OperatorToken.**pred**

Separator

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

Methods

`__init__`	Initialize self.
`ast`	
`compile`	
`match`	
`process`	
`set_expr`	
`update_input_tokens`	
`update_name`	

__init__

Separator.**__init__**(*s*, *context=None*)
Initialize self. See help(type(self)) for accurate signature.

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*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

get_n_args	
name	
node_id	
pred	

get_n_args

`Separator.get_n_args`

name

`Separator.name`

node_id

`Separator.node_id`

pred

`Separator.pred`

parenthesis

It provides Parenthesis class.

Classes

Parenthesis

Parenthesis

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

Methods

__init__	Initialize self.
ast	
match	
process	
set_expr	
update_input_tokens	

__init__

`Parenthesis.__init__(s, context=None)`
Initialize self. See help(type(self)) for accurate signature.

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)`
Initialize self. See help(type(self)) for accurate signature.

Attributes

n_args
name
node_id
opens

n_args

`Parenthesis.n_args = 0`

name

`Parenthesis.name`

node_id

`Parenthesis.node_id`

opens

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

Classes

Token

Token

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

Methods

__init__	Initialize self.
ast	
match	
process	
set_expr	
update_input_tokens	

__init__

`Token.__init__(s, context=None)`
Initialize self. See help(type(self)) for accurate signature.

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)`
Initialize self. See help(type(self)) for accurate signature.

Attributes

name
node_id

name

`Token.name`

node_id

`Token.node_id`

2.1.7.5 functions

It provides functions implementations to compile the Excel functions.

Sub-Modules:

eng	Python equivalents of engineering Excel functions.
financial	Python equivalents of financial Excel functions.
info	Python equivalents of information Excel functions.
logic	Python equivalents of logical Excel functions.
look	Python equivalents of lookup and reference Excel functions.
math	Python equivalents of math and trigonometry Excel functions.
operators	Python equivalents of Excel operators.
stat	Python equivalents of statistical Excel functions.
text	Python equivalents of text Excel functions.

eng

Python equivalents of engineering Excel functions.

financial

Python equivalents of financial Excel functions.

Functions

xirr

xirr

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

info

Python equivalents of information Excel functions.

Functions

iserr
iserror

iserr

iserr(*val*)

iserror

iserror(*val*)

Classes

IsErrArray
IsErrorArray

IsErrArray

class IsErrArray

Methods

`all`	Returns True if all elements evaluate to True.
`any`	Returns True if any of the elements of *a* 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 of *a*.
`argpartition`	Returns the indices that would partition this array.

Continued on next page

Table 47 – continued from previous page

`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.
`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`	Dot product of two arrays.
`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 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 `a.flat[n] = values[n]` for all *n* 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 *a* with each element rounded to the given number of decimals.

Continued on next page

Table 47 – continued from previous page

searchsorted	Find indices where elements of v should be inserted in a 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, and UPDATEIFCOPY, respectively.
sort	Sort an array, in-place.
squeeze	Remove single-dimensional entries from the shape of *a*.
std	Returns the standard deviation of the array elements along given axis.
sum	Return the sum of the array elements over the given axis.
swapaxes	Return a view of the array with *axis1* and *axis2* interchanged.
take	Return an array formed from the elements of *a* at the given indices.
tobytes	Construct Python bytes containing the raw data bytes in the array.
tofile	Write array to a file as text or binary (default).
tolist	Return the array as a (possibly nested) list.
tostring	Construct Python bytes containing the raw data bytes in the array.
trace	Return the sum along diagonals of the array.
transpose	Returns a view of the array with axes transposed.
var	Returns the variance of the array elements, along given axis.
view	New view of array with the same data.

all

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

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

numpy.all : equivalent function

any

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

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

Refer to *numpy.any* for full documentation.

numpy.any : equivalent function

argmax

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

Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

numpy.argmax : equivalent function

argmin

`IsErrArray.argmin(axis=None, out=None)`

Return indices of the minimum values along the given axis of *a*.

Refer to *numpy.argmin* for detailed documentation.

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.

numpy.argpartition : equivalent function

argsort

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

Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

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.

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.

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*.

Starting in NumPy 1.9, astype method now returns an error if the string dtype to cast to is not long enough in 'safe' casting mode to hold the max value of integer/float array that is being casted. Previously the casting was allowed even if the result was truncated.

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

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

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

byteswap

`IsErrArray.`**`byteswap`**(*inplace*)

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.

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

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

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

Arrays of strings are not swapped

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

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.

numpy.choose : equivalent function

clip

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

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.

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.

numpy.compress : equivalent function

conj

`IsErrArray.conj()`

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

conjugate

`IsErrArray.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

copy

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

Return a copy of the array.

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 :func:numpy.copy are very similar, but have different default values for their order= arguments.)

numpy.copy numpy.copyto

```
>>> 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.

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.

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.

numpy.diagonal : equivalent function

dot

`IsErrArray.`**`dot`**`(`*b*, *out=None*`)`

Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

numpy.dot : equivalent function

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

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.

file [str] A string naming the dump file.

dumps

`IsErrArray.`**`dumps`**`()`

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

None

fill

`IsErrArray.`**`fill`**`(`*value*`)`

Fill the array with a scalar value.

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

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

(continues on next page)

(continued from previous page)

```
>>> a
array([ 1.,  1.])
```

flatten

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

Return a copy of the array collapsed into one dimension.

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'.

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

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

```
>>> 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.

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.

```
>>> 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.

*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.

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

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

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*.

***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.

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

max

IsErrArray.**max**(*axis=None*, *out=None*)
Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

numpy.amax : equivalent function

mean

IsErrArray.**mean**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

numpy.mean : equivalent function

min

IsErrArray.**min**(*axis=None*, *out=None*, *keepdims=False*)
Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

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.

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

- {‘<’, ‘L’} - little endian
- {‘>’, ‘B’} - big endian
- {‘=’, ‘N’} - native order
- {‘|’, ‘I’} - ignore (no change to byte order)

The default value (‘S’) results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of ‘B’ or ‘b’ or ‘biggish’ are valid to specify big-endian.

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.

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 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.

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 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.

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 be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

numpy.partition : Return a parititioned copy of an array. argpartition : Indirect partition. sort : Full sort.

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

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

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

prod

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

Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

numpy.prod : equivalent function

ptp

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

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

Refer to *numpy.ptp* for full documentation.

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.

numpy.put : equivalent function

ravel

`IsErrArray.ravel([order])`

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

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.

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.

numpy.reshape : equivalent function

resize

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

Change shape and size of array in-place.

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.

None

ValueError If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

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

resize : Return a new array with the specified shape.

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.

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

```
>>> 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 has been 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.

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*

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.

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*.

None

getfield

```
>>> 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]])
>>> x
array([[  1.00000000e+000,   1.48219694e-323,   1.48219694e-323],
       [  1.48219694e-323,   1.00000000e+000,   1.48219694e-323],
       [  1.48219694e-323,   1.48219694e-323,   1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
```

(continues on next page)

(continued from previous page)

```
>>> 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, and UPDATEIFCOPY, 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 UPDATEIFCOPY 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.)

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.

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, 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);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

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

```
>>> 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
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

sort

`IsErrArray.sort` (*axis=-1*, *kind='quicksort'*, *order=None*)
Sort an array, in-place.

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

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm. Default is 'quicksort'.

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.

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

See `sort` for notes on the different sorting algorithms.

```
>>> 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([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

squeeze

`IsErrArray.squeeze` (*axis=None*)
Remove single-dimensional entries from the shape of *a*.

Refer to *numpy.squeeze* for full documentation.

numpy.squeeze : equivalent function

std

`IsErrArray.std` (*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the standard deviation of the array elements along given axis.

Refer to *numpy.std* for full documentation.

numpy.std : equivalent function

sum

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

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

Refer to *numpy.sum* for full documentation.

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.

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.

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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

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

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.

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.

tolist

`IsErrArray.tolist()`

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

none

y [list] The possibly nested list of array elements.

The array may be recreated, `a = np.array(a.tolist())`.

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

tostring

`IsErrArray.tostring(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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

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.

numpy.trace : equivalent function

transpose

`IsErrArray.`**`transpose`**`(`**axes*`)`

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

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 *a*'s *i*-th axis becomes *a.transpose()*'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)

out [ndarray] View of *a*, with axes suitably permuted.

ndarray.T : Array property returning the array transposed.

```
>>> 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]])
```

var

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

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

Refer to *numpy.var* for full documentation.

numpy.var : equivalent function

view

`IsErrArray.view(dtype=None, type=None)`

New view of array with the same data.

dtype [data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, 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, the default None results in type preservation.

`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 behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

```
>>> 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.matrixlib.defmatrix.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
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], 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[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

Attributes

T	Same as self.transpose(), except that self is returned if self.ndim < 2.
base	Base object if memory is from some other object.
ctypes	An object to simplify the interaction of the array with the ctypes module.
data	Python buffer object pointing to the start of the array's data.
dtype	Data-type of the array's elements.
flags	Information about the memory layout of the array.
flat	A 1-D iterator over the array.
imag	The imaginary part of the array.
itemsize	Length of one array element in bytes.
nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
real	The real part of the array.

Continued on next page

Table 48 – continued from previous page

shape	Tuple of array dimensions.
size	Number of elements in the array.
strides	Tuple of bytes to step in each dimension when traversing an array.

T

IsErrArray.**T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

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

base

IsErrArray.**base**

Base object if memory is from some other object.

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.

None

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

numpy.ctypeslib

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):

- 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].
- 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. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- 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.
- 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)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).
- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

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.

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
```

(continues on next page)

(continued from previous page)

```
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

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.

None

d : numpy dtype object

numpy.dtype

```
>>> 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.

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.

UPDATEIFCOPY (U) This array is a copy of some other array. When this array is deallocated, 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.

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 UPDATEIFCOPY, 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:

- UPDATEIFCOPY 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.

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

flatiter

```
>>> 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)
<type '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.

```
>>> 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.

```
>>> 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.

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

```
>>> 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.

```
>>> 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.

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

numpy.real : equivalent function

shape

IsErrArray.**shape**

Tuple of array dimensions.

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

```
>>> 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
```

size

IsErrArray.**size**

Number of elements in the array.

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

```
>>> 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.

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)`.

numpy.lib.stride_tricks.as_strided

```
>>> 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

`all`	Returns True if all elements evaluate to True.
`any`	Returns True if any of the elements of *a* 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 of *a*.
`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.
`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`	Dot product of two arrays.
`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.

Continued on next page

Table 49 – continued from previous page

`partition`	Rearranges the elements in the array in such a way that 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 `a.flat[n] = values[n]` for all *n* 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 *a* with each element rounded to the given number of decimals.
`searchsorted`	Find indices where elements of v should be inserted in a 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, and UPDATEIFCOPY, respectively.
`sort`	Sort an array, in-place.
`squeeze`	Remove single-dimensional entries from the shape of *a*.
`std`	Returns the standard deviation of the array elements along given axis.
`sum`	Return the sum of the array elements over the given axis.
`swapaxes`	Return a view of the array with *axis1* and *axis2* interchanged.
`take`	Return an array formed from the elements of *a* at the given indices.
`tobytes`	Construct Python bytes containing the raw data bytes in the array.
`tofile`	Write array to a file as text or binary (default).
`tolist`	Return the array as a (possibly nested) list.
`tostring`	Construct Python bytes containing the raw data bytes in the array.
`trace`	Return the sum along diagonals of the array.
`transpose`	Returns a view of the array with axes transposed.
`var`	Returns the variance of the array elements, along given axis.
`view`	New view of array with the same data.

all

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

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

numpy.all : equivalent function

any

`IsErrorArray.any(axis=None, out=None, keepdims=False)`
Returns True if any of the elements of *a* evaluate to True.

Refer to *numpy.any* for full documentation.

numpy.any : equivalent function

argmax

`IsErrorArray.argmax(axis=None, out=None)`
Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

numpy.argmax : equivalent function

argmin

`IsErrorArray.argmin(axis=None, out=None)`
Return indices of the minimum values along the given axis of *a*.

Refer to *numpy.argmin* for detailed documentation.

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.

numpy.argpartition : equivalent function

argsort

`IsErrorArray.argsort(axis=-1, kind='quicksort', order=None)`
Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

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.

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.

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*.

Starting in NumPy 1.9, astype method now returns an error if the string dtype to cast to is not long enough in ‘safe’ casting mode to hold the max value of integer/float array that is being casted. Previously the casting was allowed even if the result was truncated.

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

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

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

byteswap

`IsErrorArray.`**`byteswap`**`(`*inplace*`)`

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.

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

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

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

(continues on next page)

(continued from previous page)

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

Arrays of strings are not swapped

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

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.

numpy.choose : equivalent function

clip

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

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.

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.

numpy.compress : equivalent function

conj

`IsErrorArray.conj`()

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

conjugate

`IsErrorArray.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

copy

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

Return a copy of the array.

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 :func:numpy.copy are very similar, but have different default values for their order= arguments.)

numpy.copy numpy.copyto

```
>>> 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.

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.

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.

numpy.diagonal : equivalent function

dot

`IsErrorArray.`**`dot`**(*b*, *out=None*)

Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

numpy.dot : equivalent function

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

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.

file [str] A string naming the dump file.

dumps

`IsErrorArray.`**`dumps`**()

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

None

fill

`IsErrorArray.`**`fill`**(*value*)

Fill the array with a scalar value.

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

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

flatten

`IsErrorArray.flatten(order='C')`
Return a copy of the array collapsed into one dimension.

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'.

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

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

```
>>> 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.

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.

```
>>> 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.

*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.

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

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

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*.

***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.

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

max

`IsErrorArray.max` (*axis=None*, *out=None*)

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

numpy.amax : equivalent function

mean

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

Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

numpy.mean : equivalent function

min

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

Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

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.

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
- {‘<’, ‘L’} - little endian
- {‘>’, ‘B’} - big endian
- {‘=’, ‘N’} - native order
- {‘|’, ‘I’} - ignore (no change to byte order)

The default value (‘S’) results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of ‘B’ or ‘b’ or ‘biggish’ are valid to specify big-endian.

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.

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 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.

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 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.

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 be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

numpy.partition : Return a parititioned copy of an array. argpartition : Indirect partition. sort : Full sort.

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

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

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

prod

IsErrorArray.**prod**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)

Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

numpy.prod : equivalent function

ptp

IsErrorArray.**ptp**(*axis=None*, *out=None*)

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

Refer to *numpy.ptp* for full documentation.

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.

numpy.put : equivalent function

ravel

IsErrorArray.**ravel**([*order*])

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

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.

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.

numpy.reshape : equivalent function

resize

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

Change shape and size of array in-place.

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.

None

ValueError If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

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

resize : Return a new array with the specified shape.

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.

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

```
>>> 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 has been 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.

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*

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.

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*.

None

getfield

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
>>> x
array([[  1.00000000e+000,   1.48219694e-323,   1.48219694e-323],
       [  1.48219694e-323,   1.00000000e+000,   1.48219694e-323],
       [  1.48219694e-323,   1.48219694e-323,   1.00000000e+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, and UPDATEIFCOPY, 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 UPDATEIFCOPY 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.)

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.

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, 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);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

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

```
>>> 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
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
```

(continues on next page)

(continued from previous page)

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

sort

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

Sort an array, in-place.

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

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm. Default is 'quicksort'.

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.

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

See `sort` for notes on the different sorting algorithms.

```
>>> 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([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

squeeze

`IsErrorArray.`**`squeeze`**(*axis=None*)

Remove single-dimensional entries from the shape of *a*.

Refer to *numpy.squeeze* for full documentation.

numpy.squeeze : equivalent function

std

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

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

Refer to *numpy.std* for full documentation.

numpy.std : equivalent function

sum

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

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

Refer to *numpy.sum* for full documentation.

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.

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.

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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

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

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.

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.

tolist

`IsErrorArray.`**`tolist`**()

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

none

y [list] The possibly nested list of array elements.

The array may be recreated, `a = np.array(a.tolist())`.

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

tostring

`IsErrorArray.`**`tostring`**(*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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

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.

numpy.trace : equivalent function

transpose

`IsErrorArray.`**`transpose`**`(`**axes*`)`

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

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 *a*'s *i*-th axis becomes *a.transpose()*'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)

out [ndarray] View of *a*, with axes suitably permuted.

ndarray.T : Array property returning the array transposed.

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

(continues on next page)

(continued from previous page)

```
>>> 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]])
```

var

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

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

Refer to *numpy.var* for full documentation.

numpy.var : equivalent function

view

`IsErrorArray.`**`view`**(*dtype=None*, *type=None*)

New view of array with the same data.

dtype [data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, 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, the default None results in type preservation.

`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 behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> print(type(y))
<class 'numpy.matrixlib.defmatrix.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
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], 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[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Attributes

`T`	Same as self.transpose(), except that self is returned if self.ndim < 2.
`base`	Base object if memory is from some other object.
`ctypes`	An object to simplify the interaction of the array with the ctypes module.
`data`	Python buffer object pointing to the start of the array's data.
`dtype`	Data-type of the array's elements.
`flags`	Information about the memory layout of the array.
`flat`	A 1-D iterator over the array.
`imag`	The imaginary part of the array.
`itemsize`	Length of one array element in bytes.
`nbytes`	Total bytes consumed by the elements of the array.
`ndim`	Number of array dimensions.
`real`	The real part of the array.
`shape`	Tuple of array dimensions.
`size`	Number of elements in the array.
`strides`	Tuple of bytes to step in each dimension when traversing an array.

T

`IsErrorArray.T`

Same as self.transpose(), except that self is returned if self.ndim < 2.

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

base

`IsErrorArray.base`

Base object if memory is from some other object.

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.

None

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

numpy.ctypeslib

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):

- 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].
- 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. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- 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.
- 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)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).
- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

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.

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

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.

None

d : numpy dtype object

numpy.dtype

```
>>> 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.

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.

UPDATEIFCOPY (U) This array is a copy of some other array. When this array is deallocated, 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.

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 UPDATEIFCOPY, 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:

- UPDATEIFCOPY 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.

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

flatiter

```
>>> 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)
<type '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.

```
>>> 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.

```
>>> 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.

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

```
>>> 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.

```
>>> 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.

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

numpy.real : equivalent function

shape

`IsErrorArray.`**`shape`**

Tuple of array dimensions.

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

```
>>> 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):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

size

`IsErrorArray.size`

Number of elements in the array.

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

```
>>> 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.

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)`.

numpy.lib.stride_tricks.as_strided

```
>>> 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)))
```

(continues on next page)

(continued from previous page)

```
>>> 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
xif
xiferror
xiferror_otype

solve_cycle

solve_cycle(**args*)

xif

xif(*condition, x=True, y=False*)

xiferror

xiferror(*val, val_if_error*)

xiferror_otype

xiferror_otype(*val, val_if_error*)

Classes

IfArray
IfErrorArray

IfArray

class IfArray

Methods

all	Returns True if all elements evaluate to True.
any	Returns True if any of the elements of *a* 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 of *a*.
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.
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	Dot product of two arrays.
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.

Continued on next page

Table 53 – continued from previous page

`partition`	Rearranges the elements in the array in such a way that 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 `a.flat[n] = values[n]` for all *n* 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 *a* with each element rounded to the given number of decimals.
`searchsorted`	Find indices where elements of v should be inserted in a 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, and UPDATEIFCOPY, respectively.
`sort`	Sort an array, in-place.
`squeeze`	Remove single-dimensional entries from the shape of *a*.
`std`	Returns the standard deviation of the array elements along given axis.
`sum`	Return the sum of the array elements over the given axis.
`swapaxes`	Return a view of the array with *axis1* and *axis2* interchanged.
`take`	Return an array formed from the elements of *a* at the given indices.
`tobytes`	Construct Python bytes containing the raw data bytes in the array.
`tofile`	Write array to a file as text or binary (default).
`tolist`	Return the array as a (possibly nested) list.
`tostring`	Construct Python bytes containing the raw data bytes in the array.
`trace`	Return the sum along diagonals of the array.
`transpose`	Returns a view of the array with axes transposed.
`var`	Returns the variance of the array elements, along given axis.
`view`	New view of array with the same data.

all

`IfArray.`**`all`**(*axis=None*, *out=None*, *keepdims=False*)

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

numpy.all : equivalent function

any

`IfArray.any(axis=None, out=None, keepdims=False)`

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

Refer to *numpy.any* for full documentation.

numpy.any : equivalent function

argmax

`IfArray.argmax(axis=None, out=None)`

Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

numpy.argmax : equivalent function

argmin

`IfArray.argmin(axis=None, out=None)`

Return indices of the minimum values along the given axis of *a*.

Refer to *numpy.argmin* for detailed documentation.

numpy.argmin : equivalent function

argpartition

`IfArray.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.

numpy.argpartition : equivalent function

argsort

`IfArray.argsort(axis=-1, kind='quicksort', order=None)`

Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

numpy.argsort : equivalent function

astype

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

Copy of the array, cast to a specified type.

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.

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*.

Starting in NumPy 1.9, astype method now returns an error if the string dtype to cast to is not long enough in 'safe' casting mode to hold the max value of integer/float array that is being casted. Previously the casting was allowed even if the result was truncated.

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

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

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

byteswap

`IfArray.`**`byteswap`**`(`*inplace*`)`

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.

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

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

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

(continues on next page)

(continued from previous page)

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

Arrays of strings are not swapped

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

choose

`IfArray.`**`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.

numpy.choose : equivalent function

clip

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

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.

numpy.clip : equivalent function

collapse

`IfArray.`**`collapse`**(*shape*)

compress

`IfArray.`**`compress`**(*condition*, *axis=None*, *out=None*)

Return selected slices of this array along given axis.

Refer to *numpy.compress* for full documentation.

numpy.compress : equivalent function

conj

`IfArray.`**`conj`**()

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

conjugate

`IfArray.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

copy

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

Return a copy of the array.

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 :func:numpy.copy are very similar, but have different default values for their order= arguments.)

numpy.copy numpy.copyto

```
>>> 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

`IfArray.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.

numpy.cumprod : equivalent function

cumsum

`IfArray.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.

numpy.cumsum : equivalent function

diagonal

`IfArray.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.

numpy.diagonal : equivalent function

dot

`IfArray.dot(b, out=None)`

Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

numpy.dot : equivalent function

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

dump

`IfArray.dump(file)`

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

file [str] A string naming the dump file.

dumps

`IfArray.dumps()`

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

None

fill

`IfArray.fill(value)`

Fill the array with a scalar value.

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

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

flatten

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

Return a copy of the array collapsed into one dimension.

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'.

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

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

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

getfield

`IfArray.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.

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.

```
>>> 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

`IfArray.item(*args)`

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

*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.

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

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

itemset

`IfArray.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*.

***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.

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

max

`IfArray.`**`max`**(*axis=None*, *out=None*)
Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

numpy.amax : equivalent function

mean

`IfArray.`**`mean`**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

numpy.mean : equivalent function

min

`IfArray.`**`min`**(*axis=None*, *out=None*, *keepdims=False*)
Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

numpy.amin : equivalent function

newbyteorder

`IfArray.`**`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.

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
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

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

nonzero

IfArray.**nonzero**()

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

Refer to *numpy.nonzero* for full documentation.

numpy.nonzero : equivalent function

partition

IfArray.**partition**(*kth*, *axis=-1*, *kind='introselect'*, *order=None*)

Rearranges the elements in the array in such a way that 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.

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 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.

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 be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

numpy.partition : Return a parititioned copy of an array. argpartition : Indirect partition. sort : Full sort.

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

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

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

prod

IfArray.**prod**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

numpy.prod : equivalent function

ptp

IfArray.**ptp**(*axis=None*, *out=None*)
Peak to peak (maximum - minimum) value along a given axis.

Refer to *numpy.ptp* for full documentation.

numpy.ptp : equivalent function

put

IfArray.**put**(*indices*, *values*, *mode='raise'*)
Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *numpy.put* for full documentation.

numpy.put : equivalent function

ravel

IfArray.**ravel**([*order*])
Return a flattened array.

Refer to *numpy.ravel* for full documentation.

numpy.ravel : equivalent function

ndarray.flat : a flat iterator on the array.

repeat

IfArray.**repeat**(*repeats*, *axis=None*)
Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

numpy.repeat : equivalent function

reshape

IfArray.**reshape**(*shape*, *order='C'*)

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

Refer to *numpy.reshape* for full documentation.

numpy.reshape : equivalent function

resize

IfArray.**resize**(*new_shape*, *refcheck=True*)

Change shape and size of array in-place.

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.

None

ValueError If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

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

resize : Return a new array with the specified shape.

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.

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

```
>>> 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 has been referenced ...
```

Unless *refcheck* is False:

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

round

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

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

Refer to *numpy.around* for full documentation.

numpy.around : equivalent function

searchsorted

`IfArray.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*

numpy.searchsorted : equivalent function

setfield

`IfArray.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.

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*.

None

getfield

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
>>> x
array([[  1.00000000e+000,   1.48219694e-323,   1.48219694e-323],
       [  1.48219694e-323,   1.00000000e+000,   1.48219694e-323],
       [  1.48219694e-323,   1.48219694e-323,   1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

setflags

`IfArray.`**`setflags`**(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, 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 UPDATEIFCOPY 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.)

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.

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, 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);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

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

```
>>> 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
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
```

(continues on next page)

(continued from previous page)

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

sort

IfArray.**sort**(*axis=-1*, *kind='quicksort'*, *order=None*)

Sort an array, in-place.

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

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm. Default is 'quicksort'.

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.

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

See `sort` for notes on the different sorting algorithms.

```
>>> 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([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

squeeze

IfArray.**squeeze**(*axis=None*)

Remove single-dimensional entries from the shape of *a*.

Refer to *numpy.squeeze* for full documentation.

numpy.squeeze : equivalent function

std

`IfArray.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False)`
Returns the standard deviation of the array elements along given axis.

Refer to *numpy.std* for full documentation.

numpy.std : equivalent function

sum

`IfArray.sum(axis=None, dtype=None, out=None, keepdims=False)`
Return the sum of the array elements over the given axis.

Refer to *numpy.sum* for full documentation.

numpy.sum : equivalent function

swapaxes

`IfArray.swapaxes(axis1, axis2)`
Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *numpy.swapaxes* for full documentation.

numpy.swapaxes : equivalent function

take

`IfArray.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.

numpy.take : equivalent function

tobytes

`IfArray.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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

tofile

`IfArray.`**`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().

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

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.

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.

tolist

`IfArray.`**`tolist`**()

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

none

y [list] The possibly nested list of array elements.

The array may be recreated, `a = np.array(a.tolist())`.

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

tostring

`IfArray.`**`tostring`**(*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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

trace

`IfArray.`**`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.

numpy.trace : equivalent function

transpose

`IfArray.`**`transpose`**(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

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 *a*'s *i*-th axis becomes *a.transpose()*'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)

out [ndarray] View of *a*, with axes suitably permuted.

ndarray.T : Array property returning the array transposed.

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

(continues on next page)

(continued from previous page)

```
>>> 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]])
```

var

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

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

Refer to *numpy.var* for full documentation.

numpy.var : equivalent function

view

`IfArray.`**`view`**(*dtype=None*, *type=None*)

New view of array with the same data.

dtype [data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, 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, the default None results in type preservation.

`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 behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> print(type(y))
<class 'numpy.matrixlib.defmatrix.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
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], 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[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

Attributes

T	Same as self.transpose(), except that self is returned if self.ndim < 2.
base	Base object if memory is from some other object.
ctypes	An object to simplify the interaction of the array with the ctypes module.
data	Python buffer object pointing to the start of the array's data.
dtype	Data-type of the array's elements.
flags	Information about the memory layout of the array.
flat	A 1-D iterator over the array.
imag	The imaginary part of the array.
itemsize	Length of one array element in bytes.
nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
real	The real part of the array.
shape	Tuple of array dimensions.
size	Number of elements in the array.
strides	Tuple of bytes to step in each dimension when traversing an array.

T

IfArray.**T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

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

base

IfArray.**base**

Base object if memory is from some other object.

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

IfArray.**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.

None

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

numpy.ctypeslib

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):

- 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].
- 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. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- 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.
- 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)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).
- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

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.

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

data

`IfArray.`**`data`**

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

dtype

`IfArray.`**`dtype`**

Data-type of the array's elements.

None

d : numpy dtype object

numpy.dtype

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

flags

`IfArray.`**`flags`**

Information about the memory layout of the array.

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.

UPDATEIFCOPY (U) This array is a copy of some other array. When this array is deallocated, 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.

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 UPDATEIFCOPY, 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:

- UPDATEIFCOPY 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

`IfArray.`**`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.

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

flatiter

```
>>> 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)
<type '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

`IfArray.`**`imag`**

The imaginary part of the array.

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

itemsize

`IfArray.`**`itemsize`**

Length of one array element in bytes.

```
>>> 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

`IfArray.`**`nbytes`**

Total bytes consumed by the elements of the array.

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

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

ndim

`IfArray.ndim`
Number of array dimensions.

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

real

`IfArray.real`
The real part of the array.

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

numpy.real : equivalent function

shape

`IfArray.shape`
Tuple of array dimensions.

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

```
>>> 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):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

size

`IfArray.size`

Number of elements in the array.

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

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

strides

`IfArray.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.

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)`.

numpy.lib.stride_tricks.as_strided

```
>>> 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)))
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

IfErrorArray

class IfErrorArray

Methods

all	Returns True if all elements evaluate to True.
any	Returns True if any of the elements of *a* 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 of *a*.
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.
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	Dot product of two arrays.
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.

Continued on next page

Table 55 – continued from previous page

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 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 `a.flat[n] = values[n]` for all *n* 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 *a* with each element rounded to the given number of decimals.
searchsorted	Find indices where elements of v should be inserted in a 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, and UPDATEIFCOPY, respectively.
sort	Sort an array, in-place.
squeeze	Remove single-dimensional entries from the shape of *a*.
std	Returns the standard deviation of the array elements along given axis.
sum	Return the sum of the array elements over the given axis.
swapaxes	Return a view of the array with *axis1* and *axis2* interchanged.
take	Return an array formed from the elements of *a* at the given indices.
tobytes	Construct Python bytes containing the raw data bytes in the array.
tofile	Write array to a file as text or binary (default).
tolist	Return the array as a (possibly nested) list.

Continued on next page

Table 55 – continued from previous page

`tostring`	Construct Python bytes containing the raw data bytes in the array.
`trace`	Return the sum along diagonals of the array.
`transpose`	Returns a view of the array with axes transposed.
`var`	Returns the variance of the array elements, along given axis.
`view`	New view of array with the same data.

all

`IfErrorArray.all(axis=None, out=None, keepdims=False)`

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

numpy.all : equivalent function

any

`IfErrorArray.any(axis=None, out=None, keepdims=False)`

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

Refer to *numpy.any* for full documentation.

numpy.any : equivalent function

argmax

`IfErrorArray.argmax(axis=None, out=None)`

Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

numpy.argmax : equivalent function

argmin

`IfErrorArray.argmin(axis=None, out=None)`

Return indices of the minimum values along the given axis of *a*.

Refer to *numpy.argmin* for detailed documentation.

numpy.argmin : equivalent function

argpartition

`IfErrorArray.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.

numpy.argpartition : equivalent function

argsort

`IfErrorArray.`**`argsort`**(*axis=-1*, *kind='quicksort'*, *order=None*)

Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

numpy.argsort : equivalent function

astype

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

Copy of the array, cast to a specified type.

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.

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*.

Starting in NumPy 1.9, astype method now returns an error if the string dtype to cast to is not long enough in 'safe' casting mode to hold the max value of integer/float array that is being casted. Previously the casting was allowed even if the result was truncated.

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

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

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

byteswap

IfErrorArray.**byteswap**(*inplace*)

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.

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

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

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

Arrays of strings are not swapped

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

choose

IfErrorArray.**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.

numpy.choose : equivalent function

clip

IfErrorArray.**clip**(*min=None*, *max=None*, *out=None*)

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.

numpy.clip : equivalent function

collapse

IfErrorArray.**collapse**(*shape*)

compress

IfErrorArray.**compress**(*condition*, *axis=None*, *out=None*)

Return selected slices of this array along given axis.

Refer to *numpy.compress* for full documentation.

numpy.compress : equivalent function

conj

`IfErrorArray.conj()`

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

conjugate

`IfErrorArray.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

copy

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

Return a copy of the array.

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 :func:numpy.copy are very similar, but have different default values for their order= arguments.)

numpy.copy numpy.copyto

```
>>> 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

`IfErrorArray.`**`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.

numpy.cumprod : equivalent function

cumsum

`IfErrorArray.`**`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.

numpy.cumsum : equivalent function

diagonal

`IfErrorArray.`**`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.

numpy.diagonal : equivalent function

dot

`IfErrorArray.`**`dot`**`(`*b*, *out=None*`)`

Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

numpy.dot : equivalent function

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

dump

`IfErrorArray.`**`dump`**`(`*file*`)`

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

file [str] A string naming the dump file.

dumps

`IfErrorArray.dumps()`

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

None

fill

`IfErrorArray.fill(value)`

Fill the array with a scalar value.

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

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

flatten

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

Return a copy of the array collapsed into one dimension.

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'.

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

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

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

getfield

`IfErrorArray.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.

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.

```
>>> 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

`IfErrorArray.item(*args)`

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

*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.

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

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
```

(continues on next page)

(continued from previous page)

```
>>> x.item((2, 2))
3
```

itemset

IfErrorArray.**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*.

***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.

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

max

IfErrorArray.**max**(*axis=None*, *out=None*)

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

numpy.amax : equivalent function

mean

IfErrorArray.**mean**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)

Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

numpy.mean : equivalent function

min

IfErrorArray.**min**(*axis=None*, *out=None*, *keepdims=False*)

Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

numpy.amin : equivalent function

newbyteorder

IfErrorArray.**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.

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
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

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

nonzero

IfErrorArray.**nonzero**()

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

Refer to *numpy.nonzero* for full documentation.

numpy.nonzero : equivalent function

partition

IfErrorArray.**partition**(*kth*, *axis=-1*, *kind='introselect'*, *order=None*)

Rearranges the elements in the array in such a way that 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.

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 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.

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 be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

numpy.partition : Return a parititioned copy of an array. argpartition : Indirect partition. sort : Full sort.

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

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

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

prod

`IfErrorArray.`**`prod`**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)

Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

numpy.prod : equivalent function

ptp

`IfErrorArray.`**`ptp`**(*axis=None*, *out=None*)

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

Refer to *numpy.ptp* for full documentation.

numpy.ptp : equivalent function

put

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

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

Refer to *numpy.put* for full documentation.

numpy.put : equivalent function

ravel

`IfErrorArray.ravel([order])`

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

numpy.ravel : equivalent function

ndarray.flat : a flat iterator on the array.

repeat

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

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

numpy.repeat : equivalent function

reshape

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

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

Refer to *numpy.reshape* for full documentation.

numpy.reshape : equivalent function

resize

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

Change shape and size of array in-place.

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.

None

ValueError If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

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

resize : Return a new array with the specified shape.

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.

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

```
>>> 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 has been referenced ...
```

Unless *refcheck* is False:

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

round

IfErrorArray.**round**(*decimals=0*, *out=None*)

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

Refer to *numpy.around* for full documentation.

numpy.around : equivalent function

searchsorted

IfErrorArray.**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*

numpy.searchsorted : equivalent function

setfield

IfErrorArray.**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.

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*.

None

getfield

```
>>> 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]])
>>> x
array([[  1.00000000e+000,   1.48219694e-323,   1.48219694e-323],
       [  1.48219694e-323,   1.00000000e+000,   1.48219694e-323],
       [  1.48219694e-323,   1.48219694e-323,   1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

setflags

IfErrorArray.**setflags**(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, 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 UPDATEIFCOPY 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.)

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.

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, WRITE-ABLE, 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);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

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

```
>>> 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
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

sort

`IfErrorArray.sort`(*axis=-1*, *kind='quicksort'*, *order=None*)

Sort an array, in-place.

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

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm. Default is 'quicksort'.

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.

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

See `sort` for notes on the different sorting algorithms.

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

(continues on next page)

(continued from previous page)

```
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([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

squeeze

`IfErrorArray.squeeze(axis=None)`

Remove single-dimensional entries from the shape of *a*.

Refer to *numpy.squeeze* for full documentation.

numpy.squeeze : equivalent function

std

`IfErrorArray.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

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

Refer to *numpy.std* for full documentation.

numpy.std : equivalent function

sum

`IfErrorArray.sum(axis=None, dtype=None, out=None, keepdims=False)`

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

Refer to *numpy.sum* for full documentation.

numpy.sum : equivalent function

swapaxes

`IfErrorArray.swapaxes(axis1, axis2)`

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

Refer to *numpy.swapaxes* for full documentation.

numpy.swapaxes : equivalent function

take

`IfErrorArray.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.

numpy.take : equivalent function

tobytes

`IfErrorArray.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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

tofile

`IfErrorArray.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().

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

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.

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.

tolist

`IfErrorArray.tolist()`

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

none

y [list] The possibly nested list of array elements.

The array may be recreated, `a = np.array(a.tolist())`.

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

tostring

`IfErrorArray.`**`tostring`**(*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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

trace

`IfErrorArray.`**`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.

numpy.trace : equivalent function

transpose

`IfErrorArray.`**`transpose`**(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

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 *a*'s *i*-th axis becomes *a.transpose()*'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)

out [ndarray] View of *a*, with axes suitably permuted.

ndarray.T : Array property returning the array transposed.

```
>>> 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]])
```

var

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

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

Refer to *numpy.var* for full documentation.

numpy.var : equivalent function

view

`IfErrorArray.`**`view`**(*dtype=None*, *type=None*)

New view of array with the same data.

dtype [data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, 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, the default None results in type preservation.

`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 behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

```
>>> 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.matrixlib.defmatrix.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
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], 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[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Attributes

T	Same as self.transpose(), except that self is returned if self.ndim < 2.
base	Base object if memory is from some other object.
ctypes	An object to simplify the interaction of the array with the ctypes module.
data	Python buffer object pointing to the start of the array's data.
dtype	Data-type of the array's elements.
flags	Information about the memory layout of the array.
flat	A 1-D iterator over the array.
imag	The imaginary part of the array.
itemsize	Length of one array element in bytes.
nbytes	Total bytes consumed by the elements of the array.
ndim	Number of array dimensions.
real	The real part of the array.
shape	Tuple of array dimensions.
size	Number of elements in the array.
strides	Tuple of bytes to step in each dimension when traversing an array.

T

IfErrorArray.**T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

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

base

`IfErrorArray.base`

Base object if memory is from some other object.

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

`IfErrorArray.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.

None

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

numpy.ctypeslib

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):

- 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].
- 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. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- 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.
- 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)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).

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

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

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.

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

data

`IfErrorArray.`**`data`**

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

dtype

`IfErrorArray.`**`dtype`**

Data-type of the array's elements.

None

d : numpy dtype object

numpy.dtype

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

(continues on next page)

(continued from previous page)

```
>>> type(x.dtype)
<type 'numpy.dtype'>
```

flags

IfErrorArray.**flags**

Information about the memory layout of the array.

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.

UPDATEIFCOPY (U) This array is a copy of some other array. When this array is deallocated, 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.

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 UPDATEIFCOPY, 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:

- UPDATEIFCOPY 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

`IfErrorArray.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.

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

flatiter

```
>>> 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)
<type '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

`IfErrorArray.imag`

The imaginary part of the array.

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

itemsize

`IfErrorArray.itemsize`

Length of one array element in bytes.

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

(continues on next page)

(continued from previous page)

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

nbytes

`IfErrorArray.`**`nbytes`**

Total bytes consumed by the elements of the array.

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

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

ndim

`IfErrorArray.`**`ndim`**

Number of array dimensions.

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

real

`IfErrorArray.`**`real`**

The real part of the array.

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

numpy.real : equivalent function

shape

`IfErrorArray.`**`shape`**

Tuple of array dimensions.

May be used to “reshape” the array, as long as this would not require a change in the total number of elements

```
>>> 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
```

size

IfErrorArray.**size**

Number of elements in the array.

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

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

strides

IfErrorArray.**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.

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)`.

numpy.lib.stride_tricks.as_strided

```
>>> 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
```

look

Python equivalents of lookup and reference Excel functions.

Functions

xcolumn
xlookup
xmatch
xrow

xcolumn

xcolumn (*cell=None*, *ref=None*)

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*)

math

Python equivalents of math and trigonometry Excel functions.

Functions

xarabic
xarctan2
xceiling
xceiling_math
xcot
xdecimal
xeven
xfact
xfactdouble
xmod
xmround
xodd
xpower
xrandbetween
xroman
xround
xsrqtpi
xsum
xsumproduct

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*)

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=<built-in function round>*)

xsrqtpi

xsrqtpi(*number*)

xsum

xsum(**args*)

xsumproduct

xsumproduct(**args*)

operators

Python equivalents of Excel operators.

Functions

logic_input_parser	
logic_wrap	Helps call a numpy universal function (ufunc).
numeric_wrap	Helps call a numpy universal function (ufunc).

logic_input_parser

logic_input_parser(*x*, *y*)

logic_wrap

logic_wrap(*func*, ***, *input_parser=<function logic_input_parser>*, *check_error=<function get_error>*, *args_parser=<function <lambda>>*, *otype=<function <lambda>>*, *ranges=False*, ***kw*)

Helps call a numpy universal function (ufunc).

numeric_wrap

numeric_wrap(*func*, *input_parser=<function <lambda>>*, *check_error=<function get_error>*, *args_parser=<function <lambda>>*, ***, *otype=<function <lambda>>*, *ranges=False*, ***kw*)

Helps call a numpy universal function (ufunc).

Classes

OperatorArray

OperatorArray

class OperatorArray

Methods

all	Returns True if all elements evaluate to True.
any	Returns True if any of the elements of *a* 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 of *a*.
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.
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	Dot product of two arrays.
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.

Continued on next page

Table 61 – continued from previous page

`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 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 `a.flat[n] = values[n]` for all *n* 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 *a* with each element rounded to the given number of decimals.
`searchsorted`	Find indices where elements of v should be inserted in a 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, and UPDATEIFCOPY, respectively.
`sort`	Sort an array, in-place.
`squeeze`	Remove single-dimensional entries from the shape of *a*.
`std`	Returns the standard deviation of the array elements along given axis.
`sum`	Return the sum of the array elements over the given axis.
`swapaxes`	Return a view of the array with *axis1* and *axis2* interchanged.
`take`	Return an array formed from the elements of *a* at the given indices.
`tobytes`	Construct Python bytes containing the raw data bytes in the array.
`tofile`	Write array to a file as text or binary (default).
`tolist`	Return the array as a (possibly nested) list.
`tostring`	Construct Python bytes containing the raw data bytes in the array.
`trace`	Return the sum along diagonals of the array.
`transpose`	Returns a view of the array with axes transposed.
`var`	Returns the variance of the array elements, along given axis.
`view`	New view of array with the same data.

all

`OperatorArray.all(axis=None, out=None, keepdims=False)`

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

numpy.all : equivalent function

any

`OperatorArray.any(axis=None, out=None, keepdims=False)`

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

Refer to *numpy.any* for full documentation.

numpy.any : equivalent function

argmax

`OperatorArray.argmax(axis=None, out=None)`

Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

numpy.argmax : equivalent function

argmin

`OperatorArray.argmin(axis=None, out=None)`

Return indices of the minimum values along the given axis of *a*.

Refer to *numpy.argmin* for detailed documentation.

numpy.argmin : equivalent function

argpartition

`OperatorArray.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.

numpy.argpartition : equivalent function

argsort

`OperatorArray.argsort(axis=-1, kind='quicksort', order=None)`

Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

numpy.argsort : equivalent function

astype

OperatorArray.**astype**(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)
Copy of the array, cast to a specified type.

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.

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*.

Starting in NumPy 1.9, astype method now returns an error if the string dtype to cast to is not long enough in 'safe' casting mode to hold the max value of integer/float array that is being casted. Previously the casting was allowed even if the result was truncated.

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

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

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

byteswap

OperatorArray.**byteswap**(*inplace*)
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.

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

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

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

Arrays of strings are not swapped

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

choose

`OperatorArray.`**`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.

numpy.choose : equivalent function

clip

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

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.

numpy.clip : equivalent function

collapse

`OperatorArray.`**`collapse`**`(`*shape*`)`

compress

`OperatorArray.`**`compress`**`(`*condition*, *axis=None*, *out=None*`)`

Return selected slices of this array along given axis.

Refer to *numpy.compress* for full documentation.

numpy.compress : equivalent function

conj

`OperatorArray.`**`conj`**`()`

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

conjugate

`OperatorArray.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

copy

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

Return a copy of the array.

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 :func:numpy.copy are very similar, but have different default values for their order= arguments.)

numpy.copy numpy.copyto

```
>>> 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

`OperatorArray.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.

numpy.cumprod : equivalent function

cumsum

OperatorArray.**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.

numpy.cumsum : equivalent function

diagonal

OperatorArray.**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.

numpy.diagonal : equivalent function

dot

OperatorArray.**dot**(*b*, *out=None*)

Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

numpy.dot : equivalent function

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

dump

OperatorArray.**dump**(*file*)

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

file [str] A string naming the dump file.

dumps

OperatorArray.**dumps**()

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

None

fill

OperatorArray.**fill**(*value*)
Fill the array with a scalar value.

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

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

flatten

OperatorArray.**flatten**(*order='C'*)
Return a copy of the array collapsed into one dimension.

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'.

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

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

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

getfield

OperatorArray.**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.

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.

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

(continues on next page)

(continued from previous page)

```
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

OperatorArray.**item**(**args*)

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

*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.

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

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

itemset

`OperatorArray.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*.

***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.

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

max

`OperatorArray.max(axis=None, out=None)`

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

numpy.amax : equivalent function

mean

`OperatorArray.mean(axis=None, dtype=None, out=None, keepdims=False)`

Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

numpy.mean : equivalent function

min

`OperatorArray.min(axis=None, out=None, keepdims=False)`

Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

numpy.amin : equivalent function

newbyteorder

`OperatorArray.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.

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
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

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

nonzero

`OperatorArray.nonzero()`

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

Refer to *numpy.nonzero* for full documentation.

numpy.nonzero : equivalent function

partition

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

Rearranges the elements in the array in such a way that 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.

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 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.

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 be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

numpy.partition : Return a parititioned copy of an array. argpartition : Indirect partition. sort : Full sort.

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

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

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

prod

`OperatorArray.`**`prod`**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)

Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

numpy.prod : equivalent function

ptp

`OperatorArray.`**`ptp`**(*axis=None*, *out=None*)

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

Refer to *numpy.ptp* for full documentation.

numpy.ptp : equivalent function

put

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

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

Refer to *numpy.put* for full documentation.

numpy.put : equivalent function

ravel

`OperatorArray.`**`ravel`**([*order*])

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

numpy.ravel : equivalent function

ndarray.flat : a flat iterator on the array.

repeat

OperatorArray.**repeat**(*repeats*, *axis=None*)

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

numpy.repeat : equivalent function

reshape

OperatorArray.**reshape**(*shape*, *order='C'*)

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

Refer to *numpy.reshape* for full documentation.

numpy.reshape : equivalent function

resize

OperatorArray.**resize**(*new_shape*, *refcheck=True*)

Change shape and size of array in-place.

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.

None

ValueError If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

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

resize : Return a new array with the specified shape.

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.

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

```
>>> 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 has been referenced ...
```

Unless *refcheck* is False:

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

round

OperatorArray.**round**(*decimals=0*, *out=None*)

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

Refer to *numpy.around* for full documentation.

numpy.around : equivalent function

searchsorted

OperatorArray.**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*

numpy.searchsorted : equivalent function

setfield

OperatorArray.**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.

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*.

None

getfield

```
>>> 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]])
>>> x
array([[  1.00000000e+000,   1.48219694e-323,   1.48219694e-323],
       [  1.48219694e-323,   1.00000000e+000,   1.48219694e-323],
       [  1.48219694e-323,   1.48219694e-323,   1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

setflags

`OperatorArray.`**`setflags`**(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, 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 UPDATEIFCOPY 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.)

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.

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, 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);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

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

```
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
```

(continues on next page)

(continued from previous page)

```
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

sort

`OperatorArray.`**`sort`**(*axis=-1*, *kind='quicksort'*, *order=None*)

Sort an array, in-place.

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

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm. Default is 'quicksort'.

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.

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

See `sort` for notes on the different sorting algorithms.

```
>>> 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([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

squeeze

`OperatorArray.`**`squeeze`**(*axis=None*)

Remove single-dimensional entries from the shape of *a*.

Refer to *numpy.squeeze* for full documentation.

numpy.squeeze : equivalent function

std

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

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

Refer to *numpy.std* for full documentation.

numpy.std : equivalent function

sum

`OperatorArray.`**`sum`**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)

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

Refer to *numpy.sum* for full documentation.

numpy.sum : equivalent function

swapaxes

`OperatorArray.`**`swapaxes`**(*axis1*, *axis2*)

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

Refer to *numpy.swapaxes* for full documentation.

numpy.swapaxes : equivalent function

take

`OperatorArray.`**`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.

numpy.take : equivalent function

tobytes

`OperatorArray.`**`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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

tofile

`OperatorArray.`**`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().

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

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.

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.

tolist

`OperatorArray.`**`tolist`**()

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

none

y [list] The possibly nested list of array elements.

The array may be recreated, `a = np.array(a.tolist())`.

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

tostring

OperatorArray.**tostring**(*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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

trace

OperatorArray.**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.

numpy.trace : equivalent function

transpose

OperatorArray.**transpose**(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

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 *a*'s *i*-th axis becomes *a.transpose()*'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)

out [ndarray] View of *a*, with axes suitably permuted.

ndarray.T : Array property returning the array transposed.

```
>>> 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]])
```

var

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

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

Refer to *numpy.var* for full documentation.

numpy.var : equivalent function

view

`OperatorArray.`**`view`**(*dtype=None*, *type=None*)

New view of array with the same data.

dtype [data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, 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, the default None results in type preservation.

`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 behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

```
>>> 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.matrixlib.defmatrix.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
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], 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[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

Attributes

`T`	Same as self.transpose(), except that self is returned if self.ndim < 2.
`base`	Base object if memory is from some other object.
`ctypes`	An object to simplify the interaction of the array with the ctypes module.
`data`	Python buffer object pointing to the start of the array's data.
`dtype`	Data-type of the array's elements.
`flags`	Information about the memory layout of the array.
`flat`	A 1-D iterator over the array.
`imag`	The imaginary part of the array.
`itemsize`	Length of one array element in bytes.
`nbytes`	Total bytes consumed by the elements of the array.
`ndim`	Number of array dimensions.
`real`	The real part of the array.
`shape`	Tuple of array dimensions.
`size`	Number of elements in the array.
`strides`	Tuple of bytes to step in each dimension when traversing an array.

T

OperatorArray.**T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

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

base

OperatorArray.**base**

Base object if memory is from some other object.

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

`OperatorArray.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.

None

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

numpy.ctypeslib

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):

- 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].
- 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. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- 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.
- 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)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).
- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

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.

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

data

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

dtype

`OperatorArray.dtype`
Data-type of the array's elements.

None

d : numpy dtype object

numpy.dtype

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

flags

`OperatorArray.flags`
Information about the memory layout of the array.

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.

UPDATEIFCOPY (U) This array is a copy of some other array. When this array is deallocated, 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.

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 UPDATEIFCOPY, 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:

- UPDATEIFCOPY 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

`OperatorArray.`**`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.

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

flatiter

```
>>> 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)
<type '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

`OperatorArray.`**`imag`**

The imaginary part of the array.

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

itemsize

`OperatorArray.`**`itemsize`**

Length of one array element in bytes.

```
>>> 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

`OperatorArray.`**`nbytes`**

Total bytes consumed by the elements of the array.

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

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

ndim

`OperatorArray.ndim`
Number of array dimensions.

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

real

`OperatorArray.real`
The real part of the array.

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

numpy.real : equivalent function

shape

`OperatorArray.shape`
Tuple of array dimensions.

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

```
>>> 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):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

size

`OperatorArray.size`
Number of elements in the array.

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

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

strides

`OperatorArray.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.

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)`.

numpy.lib.stride_tricks.as_strided

```
>>> 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)))
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

stat

Python equivalents of statistical Excel functions.

Functions

xaverage
xmax
xmin

xaverage

xaverage(**args*)

xmax

xmax(**args*)

xmin

xmin(**args*)

text

Python equivalents of text Excel functions.

Functions

xfind
xleft
xmid

Continued on next page

Table 64 – continued from previous page

xreplace
xright

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*)

Classes

TrimArray

TrimArray

class TrimArray

Methods

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

Continued on next page

Table 66 – continued from previous page

`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.
`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`	Dot product of two arrays.
`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 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 `a.flat[n] = values[n]` for all *n* 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 *a* with each element rounded to the given number of decimals.
`searchsorted`	Find indices where elements of v should be inserted in a to maintain order.

Continued on next page

Table 66 – continued from previous page

`setfield`	Put a value into a specified place in a field defined by a data-type.
`setflags`	Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, respectively.
`sort`	Sort an array, in-place.
`squeeze`	Remove single-dimensional entries from the shape of *a*.
`std`	Returns the standard deviation of the array elements along given axis.
`sum`	Return the sum of the array elements over the given axis.
`swapaxes`	Return a view of the array with *axis1* and *axis2* interchanged.
`take`	Return an array formed from the elements of *a* at the given indices.
`tobytes`	Construct Python bytes containing the raw data bytes in the array.
`tofile`	Write array to a file as text or binary (default).
`tolist`	Return the array as a (possibly nested) list.
`tostring`	Construct Python bytes containing the raw data bytes in the array.
`trace`	Return the sum along diagonals of the array.
`transpose`	Returns a view of the array with axes transposed.
`var`	Returns the variance of the array elements, along given axis.
`view`	New view of array with the same data.

all

`TrimArray.`**`all`**`(`*axis=None*, *out=None*, *keepdims=False*`)`

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

numpy.all : equivalent function

any

`TrimArray.`**`any`**`(`*axis=None*, *out=None*, *keepdims=False*`)`

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

Refer to *numpy.any* for full documentation.

numpy.any : equivalent function

argmax

`TrimArray.`**`argmax`**`(`*axis=None*, *out=None*`)`

Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

numpy.argmax : equivalent function

argmin

`TrimArray.argmin(axis=None, out=None)`

Return indices of the minimum values along the given axis of *a*.

Refer to *numpy.argmin* for detailed documentation.

numpy.argmin : equivalent function

argpartition

`TrimArray.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.

numpy.argpartition : equivalent function

argsort

`TrimArray.argsort(axis=-1, kind='quicksort', order=None)`

Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

numpy.argsort : equivalent function

astype

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

Copy of the array, cast to a specified type.

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.

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*.

Starting in NumPy 1.9, astype method now returns an error if the string dtype to cast to is not long enough in 'safe' casting mode to hold the max value of integer/float array that is being casted. Previously the casting was allowed even if the result was truncated.

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

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

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

byteswap

`TrimArray.`**`byteswap`**`(`*inplace*`)`

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.

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

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

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

Arrays of strings are not swapped

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

choose

`TrimArray.`**`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.

numpy.choose : equivalent function

clip

`TrimArray.clip(min=None, max=None, out=None)`

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.

numpy.clip : equivalent function

collapse

`TrimArray.collapse(shape)`

compress

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

Return selected slices of this array along given axis.

Refer to *numpy.compress* for full documentation.

numpy.compress : equivalent function

conj

`TrimArray.conj()`

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

conjugate

`TrimArray.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

copy

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

Return a copy of the array.

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 :func:numpy.copy are very similar, but have different default values for their order= arguments.)

numpy.copy numpy.copyto

```
>>> 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

`TrimArray.`**`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.

numpy.cumprod : equivalent function

cumsum

`TrimArray.`**`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.

numpy.cumsum : equivalent function

diagonal

`TrimArray.`**`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.

numpy.diagonal : equivalent function

dot

`TrimArray.`**`dot`**(*b*, *out=None*)

Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

numpy.dot : equivalent function

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

dump

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

file [str] A string naming the dump file.

dumps

`TrimArray.dumps()`
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

None

fill

`TrimArray.fill(value)`
Fill the array with a scalar value.

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

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

flatten

`TrimArray.flatten(order='C')`
Return a copy of the array collapsed into one dimension.

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'.

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

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

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

getfield

`TrimArray.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.

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.

```
>>> 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

`TrimArray.item(*args)`

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

*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.

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

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

itemset

`TrimArray.`**`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*.

***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.

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

max

`TrimArray.max(axis=None, out=None)`

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

numpy.amax : equivalent function

mean

`TrimArray.mean(axis=None, dtype=None, out=None, keepdims=False)`

Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

numpy.mean : equivalent function

min

`TrimArray.min(axis=None, out=None, keepdims=False)`

Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

numpy.amin : equivalent function

newbyteorder

`TrimArray.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.

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
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

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

nonzero

`TrimArray.nonzero()`

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

Refer to *numpy.nonzero* for full documentation.

numpy.nonzero : equivalent function

partition

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

Rearranges the elements in the array in such a way that 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.

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 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.

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 be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

numpy.partition : Return a parititioned copy of an array. argpartition : Indirect partition. sort : Full sort.

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

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

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

prod

`TrimArray.prod(axis=None, dtype=None, out=None, keepdims=False)`

Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

numpy.prod : equivalent function

ptp

`TrimArray.ptp(axis=None, out=None)`

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

Refer to *numpy.ptp* for full documentation.

numpy.ptp : equivalent function

put

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

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

Refer to *numpy.put* for full documentation.

numpy.put : equivalent function

ravel

`TrimArray.ravel([order])`

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

numpy.ravel : equivalent function

ndarray.flat : a flat iterator on the array.

repeat

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

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

numpy.repeat : equivalent function

reshape

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

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

Refer to *numpy.reshape* for full documentation.

numpy.reshape : equivalent function

resize

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

Change shape and size of array in-place.

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.

None

ValueError If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

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

resize : Return a new array with the specified shape.

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.

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

```
>>> 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 has been referenced ...
```

Unless *refcheck* is False:

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

round

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

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

Refer to *numpy.around* for full documentation.

numpy.around : equivalent function

searchsorted

`TrimArray.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*

numpy.searchsorted : equivalent function

setfield

`TrimArray.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.

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*.

None

getfield

```
>>> 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]])
>>> x
array([[  1.00000000e+000,   1.48219694e-323,   1.48219694e-323],
       [  1.48219694e-323,   1.00000000e+000,   1.48219694e-323],
       [  1.48219694e-323,   1.48219694e-323,   1.00000000e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

setflags

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

Set array flags WRITEABLE, ALIGNED, and UPDATEIFCOPY, 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 UPDATEIFCOPY 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.)

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.

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, 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);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

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

```
>>> 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
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

sort

`TrimArray.sort(axis=-1, kind='quicksort', order=None)`

Sort an array, in-place.

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

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm. Default is 'quicksort'.

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.

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

See `sort` for notes on the different sorting algorithms.

```
>>> 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([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

squeeze

`TrimArray.squeeze(axis=None)`

Remove single-dimensional entries from the shape of *a*.

Refer to *numpy.squeeze* for full documentation.

numpy.squeeze : equivalent function

std

`TrimArray.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

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

Refer to *numpy.std* for full documentation.

numpy.std : equivalent function

sum

`TrimArray.sum(axis=None, dtype=None, out=None, keepdims=False)`

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

Refer to *numpy.sum* for full documentation.

numpy.sum : equivalent function

swapaxes

`TrimArray.swapaxes(axis1, axis2)`

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

Refer to *numpy.swapaxes* for full documentation.

numpy.swapaxes : equivalent function

take

`TrimArray.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.

numpy.take : equivalent function

tobytes

`TrimArray.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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

tofile

`TrimArray.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().

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

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.

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.

tolist

`TrimArray.tolist()`

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

none

y [list] The possibly nested list of array elements.

The array may be recreated, `a = np.array(a.tolist())`.

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

tostring

`TrimArray.tostring(`*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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

trace

`TrimArray.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.

numpy.trace : equivalent function

transpose

`TrimArray.transpose(*axes)`

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

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 *a*'s *i*-th axis becomes *a.transpose()*'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)

out [ndarray] View of *a*, with axes suitably permuted.

ndarray.T : Array property returning the array transposed.

```
>>> 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]])
```

var

`TrimArray.var(axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

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

Refer to *numpy.var* for full documentation.

numpy.var : equivalent function

view

`TrimArray.view(dtype=None, type=None)`
New view of array with the same data.

dtype [data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, 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, the default None results in type preservation.

`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 behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

```
>>> 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.matrixlib.defmatrix.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
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], 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[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Attributes

`T`	Same as self.transpose(), except that self is returned if self.ndim < 2.
`base`	Base object if memory is from some other object.
`ctypes`	An object to simplify the interaction of the array with the ctypes module.
`data`	Python buffer object pointing to the start of the array's data.
`dtype`	Data-type of the array's elements.
`flags`	Information about the memory layout of the array.
`flat`	A 1-D iterator over the array.
`imag`	The imaginary part of the array.
`itemsize`	Length of one array element in bytes.
`nbytes`	Total bytes consumed by the elements of the array.
`ndim`	Number of array dimensions.
`real`	The real part of the array.
`shape`	Tuple of array dimensions.
`size`	Number of elements in the array.
`strides`	Tuple of bytes to step in each dimension when traversing an array.

T

TrimArray.**T**

Same as self.transpose(), except that self is returned if self.ndim < 2.

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

base

`TrimArray.`**`base`**

Base object if memory is from some other object.

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

`TrimArray.`**`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.

None

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

numpy.ctypeslib

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):

- 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].
- 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. This base-type could be c_int, c_long, or c_longlong

depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.

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

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

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.

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

data

`TrimArray.data`

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

dtype

`TrimArray.dtype`

Data-type of the array's elements.

None

d : numpy dtype object

numpy.dtype

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

flags

`TrimArray.flags`

Information about the memory layout of the array.

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.

UPDATEIFCOPY (U) This array is a copy of some other array. When this array is deallocated, 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.

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 UPDATEIFCOPY, 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:

- UPDATEIFCOPY 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

`TrimArray.`**`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.

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

flatiter

```
>>> 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)
<type '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

`TrimArray.`**`imag`**

The imaginary part of the array.

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

itemsize

TrimArray.**itemsize**
Length of one array element in bytes.

```
>>> 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

TrimArray.**nbytes**
Total bytes consumed by the elements of the array.

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

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

ndim

TrimArray.**ndim**
Number of array dimensions.

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

real

TrimArray.**real**
The real part of the array.

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

numpy.real : equivalent function

shape

`TrimArray.shape`
Tuple of array dimensions.

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

```
>>> 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
```

size

`TrimArray.size`
Number of elements in the array.

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

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

strides

`TrimArray.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.

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)`.

numpy.lib.stride_tricks.as_strided

```
>>> 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
```

Functions

flatten	
get_error	
is_number	
not_implemented	
parse_ranges	
raise_errors	
replace_empty	
wrap_func	
wrap_ranges_func	
wrap_ufunc	Helps call a numpy universal function (ufunc).

flatten

flatten(*l, check=<function is_number>*)

get_error

get_error(**vals*)

is_number

is_number(*number*)

not_implemented

not_implemented(**args, **kwargs*)

parse_ranges

parse_ranges(**args, **kw*)

raise_errors

raise_errors(**args*)

replace_empty

replace_empty(*x, empty=0*)

wrap_func

wrap_func(*func, ranges=False*)

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=<function <lambda>>, ranges=False, **kw*)

Helps call a numpy universal function (ufunc).

Classes

Array

Array

class Array

Methods

all	Returns True if all elements evaluate to True.
any	Returns True if any of the elements of *a* 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 of *a*.
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.
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	Dot product of two arrays.
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.

Continued on next page

Table 70 – continued from previous page

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 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 `a.flat[n] = values[n]` for all *n* 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 *a* with each element rounded to the given number of decimals.
searchsorted	Find indices where elements of v should be inserted in a 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, and UPDATEIFCOPY, respectively.
sort	Sort an array, in-place.
squeeze	Remove single-dimensional entries from the shape of *a*.
std	Returns the standard deviation of the array elements along given axis.
sum	Return the sum of the array elements over the given axis.
swapaxes	Return a view of the array with *axis1* and *axis2* interchanged.
take	Return an array formed from the elements of *a* at the given indices.
tobytes	Construct Python bytes containing the raw data bytes in the array.
tofile	Write array to a file as text or binary (default).
tolist	Return the array as a (possibly nested) list.
tostring	Construct Python bytes containing the raw data bytes in the array.
trace	Return the sum along diagonals of the array.
transpose	Returns a view of the array with axes transposed.
var	Returns the variance of the array elements, along given axis.
view	New view of array with the same data.

all

Array.**all**(*axis=None*, *out=None*, *keepdims=False*)

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

numpy.all : equivalent function

any

`Array.any(axis=None, out=None, keepdims=False)`
Returns True if any of the elements of *a* evaluate to True.

Refer to *numpy.any* for full documentation.

numpy.any : equivalent function

argmax

`Array.argmax(axis=None, out=None)`
Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

numpy.argmax : equivalent function

argmin

`Array.argmin(axis=None, out=None)`
Return indices of the minimum values along the given axis of *a*.

Refer to *numpy.argmin* for detailed documentation.

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.

numpy.argpartition : equivalent function

argsort

`Array.argsort(axis=-1, kind='quicksort', order=None)`
Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

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.

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.

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*.

Starting in NumPy 1.9, astype method now returns an error if the string dtype to cast to is not long enough in 'safe' casting mode to hold the max value of integer/float array that is being casted. Previously the casting was allowed even if the result was truncated.

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

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

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

byteswap

`Array.byteswap(inplace)`

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.

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

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

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

Arrays of strings are not swapped

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

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.

numpy.choose : equivalent function

clip

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

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.

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.

numpy.compress : equivalent function

conj

`Array.`**`conj`**()

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

conjugate

`Array.conjugate()`

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

copy

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

Return a copy of the array.

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 :func:numpy.copy are very similar, but have different default values for their order= arguments.)

numpy.copy numpy.copyto

```
>>> 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.

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.

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.

numpy.diagonal : equivalent function

dot

`Array.dot(b, out=None)`

Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

numpy.dot : equivalent function

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[ 2.,  2.],
       [ 2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[ 8.,  8.],
       [ 8.,  8.]])
```

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.

file [str] A string naming the dump file.

dumps

`Array.dumps()`

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

None

fill

Array.**fill**(*value*)
Fill the array with a scalar value.

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

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

flatten

Array.**flatten**(*order='C'*)
Return a copy of the array collapsed into one dimension.

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'.

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

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

```
>>> 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.

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.

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

(continues on next page)

(continued from previous page)

```
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.

*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.

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

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.item(3)
2
>>> x.item(7)
5
>>> x.item((0, 1))
1
>>> x.item((2, 2))
3
```

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*.

***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.

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.

```
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[3, 1, 7],
       [2, 8, 3],
       [8, 5, 3]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[3, 1, 7],
       [2, 0, 3],
       [8, 5, 9]])
```

max

`Array.max(axis=None, out=None)`

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

numpy.amax : equivalent function

mean

`Array.mean(axis=None, dtype=None, out=None, keepdims=False)`

Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

numpy.mean : equivalent function

min

`Array.min(axis=None, out=None, keepdims=False)`

Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

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.

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
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

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.

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 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.

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 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.

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 be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

numpy.partition : Return a parititioned copy of an array. argpartition : Indirect partition. sort : Full sort.

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

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

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

prod

`Array.prod`(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)

Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

numpy.prod : equivalent function

ptp

`Array.ptp`(*axis=None*, *out=None*)

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

Refer to *numpy.ptp* for full documentation.

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.

numpy.put : equivalent function

ravel

`Array.ravel`([*order*])

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

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.

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.

numpy.reshape : equivalent function

resize

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

Change shape and size of array in-place.

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.

None

ValueError If *a* does not own its own data or references or views to it exist, and the data memory must be changed.

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

resize : Return a new array with the specified shape.

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.

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

```
>>> 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 has been 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.

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*

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.

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*.

None

getfield

```
>>> 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]])
>>> x
array([[  1.00000000e+000,   1.48219694e-323,   1.48219694e-323],
       [  1.48219694e-323,   1.00000000e+000,   1.48219694e-323],
       [  1.48219694e-323,   1.48219694e-323,   1.00000000e+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, and UPDATEIFCOPY, 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 UPDATEIFCOPY 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.)

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.

Array flags provide information about how the memory area used for the array is to be interpreted. There are 6 Boolean flags in use, only three of which can be changed by the user: UPDATEIFCOPY, 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);

UPDATEIFCOPY (U) this array is a copy of some other array (referenced by .base). When this array is deallocated, the base array will be updated with the contents of this array.

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

```
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
```

(continues on next page)

(continued from previous page)

```
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set UPDATEIFCOPY flag to True
```

sort

`Array.sort` (*axis=-1*, *kind='quicksort'*, *order=None*)

Sort an array, in-place.

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

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm. Default is 'quicksort'.

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.

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

See `sort` for notes on the different sorting algorithms.

```
>>> 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([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

squeeze

`Array.squeeze(axis=None)`

Remove single-dimensional entries from the shape of *a*.

Refer to *numpy.squeeze* for full documentation.

numpy.squeeze : equivalent function

std

`Array.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False)`

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

Refer to *numpy.std* for full documentation.

numpy.std : equivalent function

sum

`Array.sum(axis=None, dtype=None, out=None, keepdims=False)`

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

Refer to *numpy.sum* for full documentation.

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.

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.

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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

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

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.

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.

tolist

`Array.tolist`()

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

none

y [list] The possibly nested list of array elements.

The array may be recreated, `a = np.array(a.tolist())`.

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

tostring

`Array.tostring(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 can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

This function is a compatibility alias for tobytes. Despite its name it returns bytes not strings.

order [{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

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

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

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.

numpy.trace : equivalent function

transpose

`Array.transpose(*axes)`

Returns a view of the array with axes transposed.

For a 1-D array, this has no effect. (To change between column and row vectors, first cast the 1-D array into a matrix object.) For a 2-D array, this is the usual matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

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 *a*'s *i*-th axis becomes *a.transpose()*'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)

out [ndarray] View of *a*, with axes suitably permuted.

ndarray.T : Array property returning the array transposed.

```
>>> 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]])
```

var

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

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

Refer to *numpy.var* for full documentation.

numpy.var : equivalent function

view

`Array.view`(*dtype=None*, *type=None*)

New view of array with the same data.

dtype [data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, 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, the default None results in type preservation.

`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 behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

```
>>> 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.matrixlib.defmatrix.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
>>> print(x)
[(1, 20) (3, 4)]
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], 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[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: new type not compatible with array.
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

Attributes

`T`	Same as self.transpose(), except that self is returned if self.ndim < 2.
`base`	Base object if memory is from some other object.
`ctypes`	An object to simplify the interaction of the array with the ctypes module.
`data`	Python buffer object pointing to the start of the array's data.
`dtype`	Data-type of the array's elements.
`flags`	Information about the memory layout of the array.
`flat`	A 1-D iterator over the array.
`imag`	The imaginary part of the array.
`itemsize`	Length of one array element in bytes.
`nbytes`	Total bytes consumed by the elements of the array.
`ndim`	Number of array dimensions.
`real`	The real part of the array.
`shape`	Tuple of array dimensions.
`size`	Number of elements in the array.
`strides`	Tuple of bytes to step in each dimension when traversing an array.

T

`Array.`**`T`**

Same as self.transpose(), except that self is returned if self.ndim < 2.

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

base

`Array.`**`base`**

Base object if memory is from some other object.

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.

None

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

numpy.ctypeslib

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):

- 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].
- 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. This base-type could be c_int, c_long, or c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.
- 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.
- 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)).
- shape_as(obj): Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).
- strides_as(obj): Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

Be careful using the ctypes attribute - especially on temporary arrays or arrays constructed on the fly. For example, calling `(a+b).ctypes.data_as(ctypes.c_void_p)` returns a pointer to memory that is invalid because the array created as (a+b) is deallocated before the next Python statement. You can avoid this problem using either `c=a+b` or `ct=(a+b).ctypes`. In the latter case, ct will hold a reference to the array until ct is deleted or re-assigned.

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.

```
>>> import ctypes
>>> x
array([[0, 1],
       [2, 3]])
>>> x.ctypes.data
30439712
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long))
<ctypes.LP_c_long object at 0x01F01300>
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_long)).contents
c_long(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_longlong)).contents
c_longlong(4294967296L)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x01FFD580>
>>> x.ctypes.shape_as(ctypes.c_long)
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x01FCE620>
>>> x.ctypes.strides_as(ctypes.c_longlong)
<numpy.core._internal.c_longlong_Array_2 object at 0x01F01300>
```

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.

None

d : numpy dtype object

numpy.dtype

```
>>> 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.

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.

UPDATEIFCOPY (U) This array is a copy of some other array. When this array is deallocated, 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.

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 UPDATEIFCOPY, 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:

- UPDATEIFCOPY 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.

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

flatiter

```
>>> 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)
<type '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.

```
>>> 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.

```
>>> 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.

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

```
>>> 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.

```
>>> 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.

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

numpy.real : equivalent function

shape

`Array.shape`
Tuple of array dimensions.

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

```
>>> 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):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

size

`Array.size`

Number of elements in the array.

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

```
>>> 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.

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)`.

numpy.lib.stride_tricks.as_strided

```
>>> 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)))
```

(continues on next page)

(continued from previous page)

```
>>> 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.

numpy.reshape : equivalent function

2.1.7.6 ranges

It provides Ranges class.

Classes

Ranges

Ranges

class Ranges (*ranges=()*, *values=None*, *is_set=False*, *all_values=True*)

Methods

__init__	Initialize self.
format_range	
get_range	
push	
pushes	
set_value	
simplify	

__init__

Ranges.**__init__** (*ranges=()*, *values=None*, *is_set=False*, *all_values=True*)
Initialize self. See help(type(self)) for accurate signature.

format_range

static Ranges.**format_range**(**args*, ***kwargs*)

get_range

static Ranges.**get_range**(*format_range*, *ref*, *context=None*)

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*, *is_set=False*, *all_values=True*)
Initialize self. See help(type(self)) for accurate signature.

Attributes

input_fields
value

input_fields

Ranges.**input_fields = ('excel', 'sheet', 'n1', 'n2', 'r1', 'r2')**

value

Ranges.**value**

2.1.7.7 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

Cell	
CellWrapper	
RangesAssembler	

Cell

class Cell(*reference*, *value*, *context=None*)

Methods

__init__	Initialize self.
add	
compile	
update_inputs	

__init__

Cell.**__init__**(*reference*, *value*, *context=None*)
Initialize self. See help(type(self)) for accurate signature.

add

Cell.**add**(*dsp*, *context=None*)

compile

Cell.**compile**(*references=None*)

update_inputs

`Cell.update_inputs(references=None)`

`__init__(reference, value, context=None)`
Initialize self. See help(type(self)) for accurate signature.

Attributes

output

output

`Cell.output`

CellWrapper

`class CellWrapper(func, parse_args, parse_kwargs)`

Methods

__init__	Initialize self.
check_cycles	

__init__

`CellWrapper.__init__(func, parse_args, parse_kwargs)`
Initialize self. See help(type(self)) for accurate signature.

check_cycles

`CellWrapper.check_cycles(cycle)`

`__init__(func, parse_args, parse_kwargs)`
Initialize self. See help(type(self)) for accurate signature.

RangesAssembler

`class RangesAssembler(ref, context=None)`

Methods

__init__	Initialize self.
push	

__init__

`RangesAssembler.__init__(ref, context=None)`
Initialize self. See help(type(self)) for accurate signature.

push

`RangesAssembler.push(cell)`

`__init__(ref, context=None)`
Initialize self. See help(type(self)) for accurate signature.

Attributes

output

output

`RangesAssembler.output`

2.1.7.8 excel

It provides Excel model class.

Classes

ExcelModel

ExcelModel

`class ExcelModel`

Methods

__init__	Initialize self.
add_book	
add_cell	
add_sheet	
compile	
complete	
finish	
load	
loads	
push	
pushes	

Continued on next page

Table 83 – continued from previous page

solve_circular
write

__init__

ExcelModel.**__init__**()

Initialize self. See help(type(self)) for accurate signature.

add_book

ExcelModel.**add_book**(*book*, *context=None*, *data_only=False*)

add_cell

ExcelModel.**add_cell**(*cell*, *context*, *references=None*, *formula_references=None*, *formula_ranges=None*, *external_links=None*)

add_sheet

ExcelModel.**add_sheet**(*worksheet*, *context*)

compile

ExcelModel.**compile**(*inputs*, *outputs*)

complete

ExcelModel.**complete**()

finish

ExcelModel.**finish**(*complete=True*, *circular=False*)

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()`

write

`ExcelModel.write(books=None, solution=None)`

`__init__()`
Initialize self. See help(type(self)) for accurate signature.

`compile_class`
alias of `schedula.utils.dsp.DispatchPipe`

2.1.8 Changelog

2.1.8.1 v0.1.4 (2018-10-19)

Fix

- (tokens) #20: Improve Number regex.

2.1.8.2 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.

2.1.8.3 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.

2.1.8.4 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.

2.1.8.5 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).

2.1.8.6 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.

2.1.8.7 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.

2.1.8.8 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.

2.1.8.9 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.

2.1.8.10 v0.0.6 (2017-05-31)

Fix

- (plot): Update schedula to 0.1.12.
- (range): Sheet name without commas has this [^Wd][w.] format.

2.1.8.11 v0.0.5 (2017-05-04)

Fix

- (doc): Update schedula to 0.1.11.

2.1.8.12 v0.0.4 (2017-02-10)

Fix

- (regex): Remove deprecation warnings.

2.1.8.13 v0.0.3 (2017-02-09)

Fix

- (appveyor): Setup of lxml.
- (excel): Remove deprecation warning openpyxl.
- (requirements): Update schedula requirement 0.1.9.

2.1.8.14 v0.0.2 (2017-02-08)

Fix

- (setup): setup fails due to long description.
- (excel): Remove deprecation warning *remove_sheet* –> *remove*.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

Symbols

A

C

E

F