Glossary

along an axis

Axes are defined for arrays with more than one dimension. A 2-dimensional array has two corresponding axes: the first running vertically downwards across rows (axis 0), and the second running horizontally across columns (axis 1).

Many operation can take place along one of these axes. For example, we can sum each row of an array, in which case we operate along columns, or axis 1:

>>>

```
>>> x = np.arange(12).reshape((3,4))

>>> x
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> x.sum(axis=1)
array([ 6, 22, 38])
```

array

A homogeneous container of numerical elements. Each element in the array occupies a fixed amount of memory (hence homogeneous), and can be a numerical element of a single type (such as float, int or complex) or a combination (such as `(float, int, float)`). Each array has an associated data-type (or `dtype`), which describes the numerical type of its elements:

>>>

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

>>> x
array([ 1.,  2.,  3.])

>>> x.dtype # floating point number, 64 bits of memory per element
dtype('float64')


# More complicated data type: each array element is a combination of
# and integer and a floating point number
>>> np.array([(1, 2.0), (3, 4.0)], dtype=[('x', int), ('y', float)])
array([(1, 2.0), (3, 4.0)],
      dtype=[('x', '<i4'), ('y', '<f8')])
```

Fast element-wise operations, called `ufuncs`_, operate on arrays.

array_like

Any sequence that can be interpreted as an ndarray. This includes nested lists, tuples, scalars and existing arrays.

attribute

A property of an object that can be accessed using `obj.attribute`, e.g., `shape` is an attribute of an array:

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

BLAS

Basic Linear Algebra Subprograms (http://en.wikipedia.org/wiki/BLAS)

broadcast

NumPy can do operations on arrays whose shapes are mismatched:

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

>>> x
array([1, 2])

>>> y
array([[3],
       [4]])

>>> x + y
array([[4, 5],
       [5, 6]])
```

See `doc.broadcasting`_ for more information.

C order

See *row-major*

column-major

A way to represent items in a N-dimensional array in the 1-dimensional computer memory. In column-major order, the leftmost index “varies the fastest”: for example the array:

```
[[1, 2, 3],
 [4, 5, 6]]
```

is represented in the column-major order as:

```
[1, 4, 2, 5, 3, 6]
```

Column-major order is also known as the Fortran order, as the Fortran programming language uses it.

decorator

An operator that transforms a function. For example, a `log` decorator may be defined to print debugging information upon function execution:

```
>>> def log(f):
...     def new_logging_func(*args, **kwargs):
...         print("Logging call with parameters:", args, kwargs)
...         return f(*args, **kwargs)
...
...     return new_logging_func
```

Now, when we define a function, we can “decorate” it using `log`:

```
>>> @log
... def add(a, b):
...     return a + b
```

Calling `add` then yields:

```
>>> add(1, 2)
Logging call with parameters: (1, 2) {}
3
```

dictionary

Resembling a language dictionary, which provides a mapping between words and descriptions thereof, a Python dictionary is a mapping between two objects:

```
>>> x = {1: 'one', 'two': [1, 2]}
```

Here, *x* is a dictionary mapping keys to values, in this case the integer 1 to the string “one”, and the string “two” to the list `[1, 2]`. The values may be accessed using their corresponding keys:

```
>>> x[1]
'one'

>>> x['two']
[1, 2]
```

Note that dictionaries are not stored in any specific order. Also, most mutable (see *immutable* below) objects, such as lists, may not be used as keys.

For more information on dictionaries, read the Python tutorial (http://docs.python.org/tut).

Fortran order

See *column-major*

flattened

Collapsed to a one-dimensional array. See `ndarray.flatten`_ for details.

immutable

An object that cannot be modified after execution is called immutable. Two common examples are strings and tuples.

instance

A class definition gives the blueprint for constructing an object:

```
>>> class House(object):
...     wall_colour = 'white'
```

Yet, we have to *build* a house before it exists:

```
>>> h = House() # build a house
```

Now, `h` is called a `House` instance. An instance is therefore a specific realisation of a class.

iterable

A sequence that allows “walking” (iterating) over items, typically using a loop such as:

```
>>> x = [1, 2, 3]
>>> [item**2 for item in x]
[1, 4, 9]
```

It is often used in combination with `enumerate`::

```
>>> keys = ['a','b','c']
>>> for n, k in enumerate(keys):
...     print("Key %d: %s" % (n, k))
...
Key 0: a
Key 1: b
Key 2: c
```

list

A Python container that can hold any number of objects or items. The items do not have to be of the same type, and can even be lists themselves:

```
>>> x = [2, 2.0, "two", [2, 2.0]]
```

The list *x* contains 4 items, each which can be accessed individually:

```
>>> x[2] # the string 'two'
'two'

>>> x[3] # a list, containing an integer 2 and a float 2.0
[2, 2.0]
```

It is also possible to select more than one item at a time, using *slicing*:

```
>>> x[0:2] # or, equivalently, x[:2]
[2, 2.0]
```

In code, arrays are often conveniently expressed as nested lists:

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

For more information, read the section on lists in the Python tutorial (http://docs.python.org/tut). For a mapping type (key-value), see *dictionary*.

mask

A boolean array, used to select only certain elements for an operation:

```
>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])

>>> mask = (x > 2)
>>> mask
array([False, False, False, True,  True], dtype=bool)

>>> x[mask] = -1
>>> x
array([ 0,  1,  2,  -1, -1])
```

masked array

Array that suppressed values indicated by a mask:

```
>>> x = np.ma.masked_array([np.nan, 2, np.nan], [True, False, True])
>>> x
masked_array(data = [-- 2.0 --],
             mask = [ True False  True],
       fill_value = 1e+20)


>>> x + [1, 2, 3]
masked_array(data = [-- 4.0 --],
             mask = [ True False  True],
       fill_value = 1e+20)
```

Masked arrays are often used when operating on arrays containing missing or invalid entries.

matrix

A 2-dimensional ndarray that preserves its two-dimensional nature throughout operations. It has certain special operations, such as `*` (matrix multiplication) and `**` (matrix power), defined:

```
>>> x = np.mat([[1, 2], [3, 4]])
>>> x
matrix([[1, 2],
        [3, 4]])

>>> x**2
matrix([[ 7, 10],
      [15, 22]])
```

method

A function associated with an object. For example, each ndarray has a method called `repeat`:

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

ndarray

See *array*.

record array

An \`ndarray\`_ with \`structured data type\`_ which has been subclassed as np.recarray and whose dtype is of type np.record, making the fields of its data type to be accessible by attribute.

reference

If `a` is a reference to `b`, then `(a is b) == True`. Therefore, `a` and `b` are different names for the same Python object.

row-major

A way to represent items in a N-dimensional array in the 1-dimensional computer memory. In row-major order, the rightmost index “varies the fastest”: for example the array:

```
[[1, 2, 3],
 [4, 5, 6]]
```

is represented in the row-major order as:

```
[1, 2, 3, 4, 5, 6]
```

Row-major order is also known as the C order, as the C programming language uses it. New NumPy arrays are by default in row-major order.

self

Often seen in method signatures, `self` refers to the instance of the associated class. For example:

```
>>> class Paintbrush(object):
...     color = 'blue'
...
...     def paint(self):
...         print("Painting the city %s!" % self.color)
...
>>> p = Paintbrush()
>>> p.color = 'red'
>>> p.paint() # self refers to 'p'
Painting the city red!
```

slice

Used to select only certain elements from a sequence:

```
>>> x = range(5)
>>> x
[0, 1, 2, 3, 4]

>>> x[1:3] # slice from 1 to 3 (excluding 3 itself)
[1, 2]

>>> x[1:5:2] # slice from 1 to 5, but skipping every second element
[1, 3]

>>> x[::-1] # slice a sequence in reverse
[4, 3, 2, 1, 0]
```

Arrays may have more than one dimension, each which can be sliced individually:

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

>>> x[:, 1]
array([2, 4])
```

structured data type

A data type composed of other datatypes

tuple

A sequence that may contain a variable number of types of any kind. A tuple is immutable, i.e., once constructed it cannot be changed. Similar to a list, it can be indexed and sliced:

```
>>> x = (1, 'one', [1, 2])
>>> x
(1, 'one', [1, 2])

>>> x[0]
1

>>> x[:2]
(1, 'one')
```

A useful concept is “tuple unpacking”, which allows variables to be assigned to the contents of a tuple:

```
>>> x, y = (1, 2)
>>> x, y = 1, 2
```

This is often used when a function returns multiple values:

```
>>> def return_many():
...     return 1, 'alpha', None
```

```
>>> a, b, c = return_many()
>>> a, b, c
(1, 'alpha', None)
```

```
>>> a
1
>>> b
'alpha'
```

ufunc

Universal function. A fast element-wise array operation. Examples include `add`, `sin` and `logical_or`.

view

An array that does not own its data, but refers to another array’s data instead. For example, we may create a view that only shows every second element of another array:

```
>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])

>>> y = x[::2]
>>> y
array([0, 2, 4])

>>> x[0] = 3 # changing x changes y as well, since y is a view on x
>>> y
array([3, 2, 4])
```

wrapper

Python is a high-level (highly abstracted, or English-like) language. This abstraction comes at a price in execution speed, and sometimes it becomes necessary to use lower level languages to do fast computations. A wrapper is code that provides a bridge between high and the low level languages, allowing, e.g., Python to execute code written in C or Fortran.

Examples include ctypes, SWIG and Cython (which wraps C and C++) and f2py (which wraps Fortran).

Jargon

Table Of Contents (contents.html)

Previous topic

NumPy License (license.html)