Numpy

Data Types

bool_ Boolean (True or False) stored as a byte

int_ Default integer type (same as C long; normally either int64 or int32)

intc Identical to C int (normally int32 or int64)

intp Integer used for indexing (same as C ssize_t; normally either int32 or int64)

int8 Byte (-128 to 127)

int16 Integer (-32768 to 32767)

int32 Integer (-2147483648 to 2147483647)

int64 Integer (-9223372036854775808 to 9223372036854775807)

uint8 Unsigned integer (0 to 255)

uint16 Unsigned integer (0 to 65535)

uint32 Unsigned integer (0 to 4294967295)

uint64 Unsigned integer (0 to 18446744073709551615)

float_ Shorthand for float64

float16 Half precision float: sign bit, 5 bits exponent, 10 bits mantissa

float32 Single precision float: sign bit, 8 bits exponent, 23 bits mantissa

float64 Double precision float: sign bit, 11 bits exponent, 52 bits mantissa

complex_ Shorthand for complex128

complex64 Complex number, represented by two 32-bit floats (real and imaginary components)

complex128 Complex number, represented by two 64-bit floats (real and imaginary components)

In [1]:
```
import numpy as np
```

In [3]:
```
#using different Datatypes
x=np.float32(1.0)
x
```

Out[3]: 1.0

In [4]:
```
y=np.int_([1,2,3])
y
```

Out[4]: array([1, 2, 3])

In [5]:
```
z = np.arange(3, dtype=np.uint8)
z
```

Out[5]: array([0, 1, 2], dtype=uint8)

In [6]:
```
np.array([1, 2, 3], dtype='f')
```
Out[6]:
```
array([1., 2., 3.], dtype=float32)
```

In [7]:
```
#convert into other datatype
z.astype(float)
```
Out[7]:
```
array([0., 1., 2.])
```

In [8]:
```
np.int8(z)
```
Out[8]:
```
array([0, 1, 2], dtype=int8)
```

In [9]:
```
z.dtype
```
Out[9]:
```
dtype('uint8')
```

In [10]:
```
d = np.dtype(int)
d
```
Out[10]:
```
dtype('int64')
```

In [11]:
```
#check for the datatype
np.issubdtype(d, np.integer)
```
Out[11]:
```
True
```

In [12]:
```
np.issubdtype(d, np.floating)
```
Out[12]:
```
False
```

In [13]:
```
#Creating arrays using numpy
x = np.array([2,3,1,0])
x
```
Out[13]:
```
array([2, 3, 1, 0])
```

In [14]:
```
x = np.array([[1,2.0],[0,0],(1+1j,3.)])
x
```
Out[14]:
```
array([[1.+0.j, 2.+0.j],
       [0.+0.j, 0.+0.j],
       [1.+1.j, 3.+0.j]])
```

In [15]:
```
np.zeros((2, 3))
```
Out[15]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

In [16]:
```
np.arange(10)
```
Out[16]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [17]:
```
np.arange(1,5,.5)
```
Out[17]:
```
array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

In [20]:
```
np.linspace(1., 5., 6)
```
Out[20]:
```
array([1. , 1.8, 2.6, 3.4, 4.2, 5. ])
```

In [22]:
```python
np.indices((3,3))
```

Out[22]:
```
array([[[0, 0, 0],
        [1, 1, 1],
        [2, 2, 2]],

       [[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])
```

indices() will create a set of arrays (stacked as a one-higher dimensioned array), one per dimension with each representing variation in that dimension

In [23]:
```python
from io import StringIO
```

Importing data with genfromtxt

NumPy provides several functions to create arrays from tabular data. We focus here on the **genfromtxt** function.

In a nutshell, genfromtxt runs two main loops. The first loop converts each line of the file in a sequence of strings. The second loop converts each string to the appropriate data type. This mechanism is slower than a single loop, but gives more flexibility. In particular, genfromtxt is able to take missing data into account, when other faster and simpler functions like loadtxt cannot.

In [24]:
```python
data = u"1, 2, 3\n4, 5, 6"
```

In [26]:
```python
np.genfromtxt(StringIO(data), delimiter=",")
```

Out[26]:
```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

In [27]:
```python
data = u"  1  2  3\n  4  5 67\n890123  4"
np.genfromtxt(StringIO(data), delimiter=3)
```

Out[27]:
```
array([[  1.,   2.,   3.],
       [  4.,   5.,  67.],
       [890., 123.,   4.]])
```

In [28]:
```python
data = u"1, abc , 2\n 3, xxx, 4"
np.genfromtxt(StringIO(data), delimiter=",", dtype="|U5", autostrip=True)
```

Out[28]:
```
array([['1', 'abc', '2'],
       ['3', 'xxx', '4']], dtype='<U5')
```

The autostrip argument

By default, when a line is decomposed into a series of strings, the individual entries are not stripped of leading nor trailing white spaces. This behavior can be overwritten by setting the optional argument **autostrip** to a value of True:

In [29]:
```
data = u"""#
... # Skip me !
... # Skip me too !
... 1, 2
... 3, 4
... 5, 6 #This is the third line of the data
... 7, 8
... # And here comes the last line
... 9, 0
... """
```

In [30]:
```
np.genfromtxt(StringIO(data), comments="#", delimiter=",")
```

Out[30]:
```
array([[1., 2.],
       [3., 4.],
       [5., 6.],
       [7., 8.],
       [9., 0.]])
```

The comments argument

The optional argument comments is used to define a character string that marks the beginning of a comment. By default, genfromtxt assumes comments='#'. The comment marker may occur anywhere on the line. Any character present after the comment marker(s) is simply ignored:

In [31]:
```
data = u"\n".join(str(i) for i in range(10))
np.genfromtxt(StringIO(data),)
```

Out[31]:
```
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

In [32]:
```
np.genfromtxt(StringIO(data),skip_header=3, skip_footer=5)
```

Out[32]:
```
array([3., 4.])
```

Skipping lines and choosing columns

The skip_header and skip_footer arguments

The presence of a header in the file can hinder data processing. In that case, we need to use the skip_header optional argument. The values of this argument must be an integer which corresponds to the number of lines to skip at the beginning of the file, before any other action is performed. Similarly, we can skip the last n lines of the file by using the skip_footer attribute and giving it a value of n:

Bracket Indexing and Selection

The simplest way to pick one or some elements of an array looks very similar to python lists:

In [34]:
```
x = np.arange(10)
>>> x[2]
```

Out[34]:
```
2
```

In [35]:
```
x.shape = (2,5) # now x is 2-dimensional
>>> x[1,3]
```

Out[35]:
```
8
```

```
In [36]: x[0]
```

```
Out[36]: array([0, 1, 2, 3, 4])
```

```
In [37]:  x = np.arange(10,1,-1)
>>> x
```

```
Out[37]: array([10,  9,  8,  7,  6,  5,  4,  3,  2])
```

```
In [38]: x[np.array([3, 3, 1, 8])]
```

```
Out[38]: array([7, 7, 9, 2])
```

Broadcasting

Numpy arrays differ from a normal Python list because of their ability to broadcast:

```
In [42]: x[1:3]=20
x
```

```
Out[42]: array([10, 20, 20,  7,  6,  5,  4,  3,  2])
```

```
In [43]: slice_arr = x[0:4]
slice_arr
```

```
Out[43]: array([10, 20, 20,  7])
```

```
In [44]: slice_arr[:]=20
```

```
In [45]: x
```

```
Out[45]: array([20, 20, 20, 20,  6,  5,  4,  3,  2])
```

Structured arrays

Introduction

Structured arrays are ndarrays whose datatype is a composition of simpler datatypes organized as a sequence of named fields. For example,

```
In [49]: x = np.array([('Rex', 9, 81.0), ('Fido', 3, 27.0)],
...              dtype=[('name', 'U10'), ('age', 'i4'), ('weight', 'f
4')])
```

```
In [51]: x
```

```
Out[51]: array([('Rex', 9, 81.), ('Fido', 3, 27.)],
      dtype=[('name', '<U10'), ('age', '<i4'), ('weight', '<f4')])
```

```
In [52]: x[1]
```

```
Out[52]: ('Fido', 3, 27.)
```

```
In [53]: x['age']
```

```
Out[53]: array([9, 3], dtype=int32)
```

In [54]:
```
x['age']=5
x
```

Out[54]:
```
array([('Rex', 5, 81.), ('Fido', 5, 27.)],
      dtype=[('name', '<U10'), ('age', '<i4'), ('weight', '<f4')])
```

In [55]:
```
np.dtype([('x', 'f4'), ('y', np.float32), ('z', 'f4', (2,2))])
```

Out[55]:
```
dtype([('x', '<f4'), ('y', '<f4'), ('z', '<f4', (2, 2))])
```

In [56]:
```
np.dtype([('x', 'f4'),('', 'i4'),('z', 'i8')])
```

Out[56]:
```
dtype([('x', '<f4'), ('f1', '<i4'), ('z', '<i8')])
```

In [57]:
```
np.dtype('i8,f4,S3')
```

Out[57]:
```
dtype([('f0', '<i8'), ('f1', '<f4'), ('f2', 'S3')])
```

In [58]:
```
np.dtype('3int8, float32, (2,3)float64')
```

Out[58]:
```
dtype([('f0', 'i1', (3,)), ('f1', '<f4'), ('f2', '<f8', (2, 3))])
```

Subclassing ndarray

Introduction

Subclassing ndarray is relatively simple, but it has some complications compared to other Python objects. On this page we explain the machinery that allows you to subclass ndarray, and the implications for implementing a subclass.

In [61]:
```
class C(np.ndarray): pass
arr = np.zeros((3,))
```

In [63]:
```
c_arr = arr.view(C)
type(c_arr)
```

Out[63]:
```
__main__.C
```

In [64]:
```
v = c_arr[1:]
```

In [65]:
```
type(v)
```

Out[65]:
```
__main__.C
```