0 | Python Introduction

Numpy

Array

The basic object in NumPy is the array, which is conceptually similar to a matrix. The NumPy array class is called ndarray (for “n-dimensional array”). The simplest way to explicitly create a 1-D ndarray is to define a list, then cast that list as an ndarray with NumPy’s array() function.

```
>>> import numpy as np
# Create a 1-D array by passing a list into NumPy's array() function.
>>> np.array([8, 4, 6, 0, 2])
array([8, 4, 6, 0, 2])
# The string representation has no commas or an array() label.
>>> print(np.array([1, 3, 5, 7, 9]))
[1 3 5 7 9]
```

The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

You can also specify the type of the elements in a array.

```
>>> import numpy as np
>>> a = np.array([2,3,4],'d')
>>> a
array([ 2.,  3.,  4.])
>>> a.dtype
dtype('float64')
>>> b = np.array([1.2, 3.5, 5.1],'i')
>>> b
array([1, 3, 5], dtype=int32)
>>> b.dtype
dtype('int32')
```

Addition/substraction of Arrays

NumPy arrays behave differently with respect to the binary arithmetic operators + and * than Python lists do. For lists, + concatenates two lists and * replicates a list by a scalar amount (strings also behave this way).

```
# Addition concatenates lists together.
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
# Multiplication concatenates a list with itself a given number of times.
>>> [1, 2, 3] * 4
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

NumPy arrays act like mathematical vectors and matrices: + and * perform component-wise addition or multiplication.

```
>>> x, y = np.array([1, 2, 3]), np.array([4, 5, 6])
# Addition or multiplication by a scalar acts on each element of the array.
>>> x + 10
# Add 10 to each entry of x.
array([11, 12, 13])
>>> x * 4
# Multiply each entry of x by 4.
array([ 4, 8, 12])
# Add two arrays together (component-wise).
>>> x + y
array([5, 7, 9])
# Multiply two arrays together (component-wise).
>>> x * y
array([ 4, 10, 18])
```

Sequences

To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

When arange is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function linspace that receives as an argument the number of elements that we want, instead of the step:

```
>>> from numpy import pi
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )        # useful to evaluate function
>>> f = np.sin(x)
```

Two other useful commands are **numpy.ones** and **numpy.zeros**.

```
 >>> import numpy as np
>>> z = np.zeros(10,'d')
>>> print(z)
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
>>> w = np.ones(10,'d')
>>> print(w)
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

Indexing

Indexing for a 1-D NumPy array uses the slicing syntax x[start:stop:step]. If there is no colon, a single entry of that dimension is accessed. With a colon, a range of values is accessed.

```
# Make an array of the integers from 0 to 10 (exclusive).
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# Access elements of the array with slicing syntax.
>>> x[3]
# The element at index 3.
3
>>> x[:3]
# Everything up to index 3 (exclusive).
array([0, 1, 2])
>>> x[3:]
# Everything from index 3 on.
array([3, 4, 5, 6, 7, 8, 9])
>>> x[3:8]
# The elements from index 3 to 8.
array([3, 4, 5, 6, 7])
```

So-called fancy indexing is a second way to access or change the elements of an array. Instead of using slicing syntax, provide either an array of indices or an array of boolean values (called a mask) to extract specific elements.

```
>>> x = np.arange(0, 50, 10)
>>> x
array([ 0, 10, 20, 30, 40])
# The integers from 0 to 50 by tens.
# An array of integers extracts the entries of 'x' at the given indices.
>>> index = np.array([3, 1, 4])
# Get the 3rd, 1st, and 4th elements.
>>> x[index]
# Same as np.array([x[i] for i in index]).
array([30, 10, 40])
# A boolean array extracts the elements of 'x' at the same places as 'True'.
>>> mask = np.array([True, False, False, True, False])
>>> x[mask]
# Get the 0th and 3rd entries.
array([ 0, 30])
```

Fancy indexing is especially useful for extracting or changing the values of an array that meet some sort of criterion. Use comparison operators like < and == to create masks.

```
>>> y = np.arange(10, 20, 2)
>>> y
array([10, 12, 14, 16, 18])
# Every other integers from 10 to 20.
# Extract the values of 'y' larger than 15.
>>> mask = y > 15
# Same as np.array([i > 15 for i in y]).
>>> mask
array([False, False, False, True, True], dtype=bool)
>>> y[mask]
# Same as y[y > 15]
array([16, 18])
# Change the values of 'y' that are larger than 15 to 100.
>>> y[mask] = 100
>>> print(y)
[10 12 14 100 100]
```

Matplotlib

Matploblib is a collection of command style functions that make matplotlib work like MATLAB. Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.
Generating visualizations with pyplot is very quick:

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
plt.show()
```

If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates a plotting several lines with different format styles in one command using arrays.

```
import numpy as np

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, np.sin(t), 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

Last example shows how to save the figure:

```python
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

# evenly 100 sampled times
t = np.linspace(0.0, 5.0, 100)

plt.figure(1)
plt.plot(t, f(t), 'k')
plt.savefig('figure.eps')           # save fhe figure
plt.show()                          # plot the figure
```