Python Basics

Thomas Johnson

May 2, 2022

In this supplementary file to the main file we record some basic features of the python programming language. We will ultimately need these in order to do plots of various derivative products.

1 File handling

This section concerns reading from and writing to (text) files in python.

1.1 Input

In python to open a (text) file for reading the synatx is as follows[1]:

open('path directory of file', 'r')

This essentially returns an input stream like object, so we can assign it a name:

f = open('path directory of file', 'r')

There are then 3 main methods for reading from the input stream:

f.read() // reads whole file
f.read(x) // reads first x characters
f.readline() // reads first line of file (delineated by newline)

One then outputs the text using the print function.
Note that by calling each of these methods what is read is 'eaten' from the input stream e.g.

f.read()
print(f.readline()) // empty line

The file itself is left unchanged however precisely because it is only open for reading.
When finished one should always close the file using the close() method. This ensures the file is in the correct state. In particular one will not be able to change the file until it has been closed again.

[1] If the file is in the same directory as the current solution then one can simply type the file name e.g. 'file.txt'.

1.2 Output

To write to a file we need to pass in 'a' or 'w' as a parameter to the open function. The former, short for append, allows to write to the end of the file whilst the latter, short for write, allows to overwrite the file. To actually do the writing one uses the write() method:

f = open('path directory of file', 'w')
f.write('Hello, world!') // overwrites file with Hello, world!

If a file with the current name doesn't already exist, the then calling open with either of those parameters will simply create a new file of that name in the current directory.

To avoid overwriting a file by mistake one can pass in 'x', short for create, as a parameter instead. This will return an error if a file of the given name already exists, and otherwise create a file of that name.

2 The Numpy module

The numpy module is a library used for working with arrays. It stands for numerical python.

The main objects in this library as ndarrays. They are much faster for working with than the standard python list objects[2] (e.g. for looping over) for 2 main reasons:

1. They are represented in memory as one continuous block ("locality of reference")
2. They are optimised to work with the latest CPU architechture (vectorisation)

They also come equipped with many supporting functions designed for numerical work.

The numpy module first needs to be downloaded and installed. It then needs to be imported. Typically one uses the alias np:

import numpy as np

2.1 Arrays

As mentioned, when using numpy one works with arrays, objects of type numpy.ndarray (n-dim array). One should think of an n-dimensional array as representing an n-dimensional tensor. In particular, an n-dimensional is built out of (n-1)-dimensional arrays, thought of as elements:

arr = np.array([1,2,4]) // 1-d array built out of scalars, 0-d arrays
arr = np.array([1,2], [3,4]) // 2-d array built out of 1-d arrays

In particular a 1-d array represents a vector, whilst a 2-d array can be thought as a matrix.

To ascertain the dimension of an array one calls the ndim attribute[3]. In particular, ndim is the number of subarrays in the array.

To access the elements (i.e. the subarrays) of an array one uses the familiar subscript notation:

[2]In fact arrays are mainly written in C/C++ as opposed to lists, which are written in python.

[3]In python a class attribute is the same as a class data member in C++. In particular one accesses the attribute of a class object using the familiar full stop syntax.

arr[i] // i'th subarray of arr

This chains:

arr[i, j] // j'th subarray of the i'th subarray of arr

In particular, to be consistent with matrix notation, for a 2d array the i'th subarray represents the i'th row of the corresponding matrix.

2.2 Iterating

To iterate over the elements of an array one uses colon syntax:

arr[i:j] // a[i],...,a[j-1]
arr[i:] // a[i],..., a[ndim-1]

Note this procedure is called *slicing* in python parlance.
One can also iterate backwards by passing in negative indices, aswell as proscribing a step value:

arr[i:j:m] // arr[i], a[i+m], ...

2.3 Numpy data types

One can check/convert data types using the dtype and astype methods of the ndarray class (useful if e.g. want to convert floats to ints). One can also declare the data type as a parameter when instantiating (e.g. one might wish for an number to be stored as a string).

2.4 Copy vs view

One creates a copy of an ndarray object using the copy method whilst the view method creates a reference (in particular a copy owns the data whereas a view does not).
The base attribute of the ndarray class can be used to check if one is working with a copy or a view. In the former case it returns 'None' whilst in the latter it returns the original object.

2.5 Shape and reshape

The attribute shape of the ndarray class returns the dimension of the array aswell as the dimensions of the subarrays which make up the array:

a = np.array([1,2,3], [1,2,3])
a.shape // (2,3)

The reshape method can also be used to change the shape of an array, provided certain conditions are met. Note it returns a view, not a copy.

2.6 for-range loops

One can iterate over the elements of an array using a basic for loop[4]:

```
a = np.array([1,2])
for x in a:
print(x) // 1,2
```

Of course for higher dimensional arrays this will return the subarray. To get the elements of the subarrays one can iterate again:

```
a = np.array([1,2], [3,4])
for x in a:
for y in x: print(y) // 1,2 3,4
```

This clearly becomes cumbersome in higher dimensions and so for this reason numpy provides the function nditer:

```
for x in np.nditer(a):
print(x) // 1,2,3,4
```

One can also use : to specify the start, end, step sizes etc.

If one wants to return the index at the same time then one can use the ndunemerate function:

```
for idx, x in np.ndunemarte(a):
print(idx, x) // [0,0], 1 etc.
```

Note in particular that the indexes are returned as an array.

2.7 Joining arrays

To join two arrays one uses the concatenate function in np. This results in an array whose elements are the unions of the elements in the two joining arrays.

The stack, hstack and vstack functions are used for joining along a particular axis.

The reverse operation is given by the splitting function.

2.8 Search and sort

The where function returns a tuple representing the indexes where an array takes a particular value:

```
x = np.where(arr==4)
```

On the otherhand the sort function sorts an array. Note it in fact returns a copy, so that the original is not effected.

[4]Remember in python one needs to respect indentation!

2.9 Filtering

Filtering is the process of getting elements out of an existing array and sorting them. It is done in numpy using a boolean index list:

```
arr = np.array([1,2,3,4]
x = [True, False, True, True]
newarr = arr[x] // = [1,3,4]
```

Typically, filtering is done based on certain conditions instead of a given index:

```
filterarr = []
for x in arr:
if x¿10: filterarr.append(True)
newarr = arr[filterarr]
```

This is such a common operation that in fact numpy provides a remarkably simple syntax for doing it:

```
filterarr = arr ¿ 10.
```

2.10 ufuncs

A ufunc, short for *universal function*, is one that operates on an ndarray object. It permits *vectorisation* a procedure that is faster than iteration, essentially because modern CPU's are optimised for such a task. There are ufuncts provided in np for doing many tasks such as adding, summing, applying certain mathematical functions etc:

```
np.add(arr1, arr2) // returns array with a[i]=a1[i]+a2[i]
np.sum(arr) // sums elements
np.prod(arr) // takes products of elements
np.sin(arr) // returns array with a[i]=sin (a[i])
```

Remarkably, np allows easily making a ufunc from a given user defined function using np.frompyfunc(). This function takes in the name of the function, the number of arrays it acts on the number of array it outputs:

```
def myadd(x,y):
return x+y
myadd = np.frompyfunc(myadd, 2, 1)
myadd(arr1, arr2)
sames as np.add(arr1, arr2)
```

3 The Matplotlib module

In this section we discuss the matplotlib module. This is a low level graph plotting library in python. It is open source, with the source code found on github. It is mostly written in python.

Most of its utilities can be found in the pyplot submodule. This is typically imported under the plt alias:

```
import matplotlib.pyplot as plt
```

3.1 Plotting

To make plots we make use of the numpy module:

```
xpts = np.array([0,6])
ypts = np.array([0,250])
plt.plot(x,y) // draws succesive lines between points (x[i], y[i]) and (x[i+1], y[i+1])
plt.show() // outputs plot to screen
```

In particular to draw a detailed graph of e^x over 0 to 100:

```
xpts = np.linspace(0, 100, 1000) // np.linspace(a,b,N) returns 1-d array of N elements with a[i]
= a+ i(b-a)/N
ypts = np.exp(xpts)
plt.plot(x,y)
plt.show()
```

There are also multiple ways of formating the plot e.g. the format of the line, the colours etc.

One can also draw multiple graphs on the same plot simply by calling the plot function multiple times:

```
xpts = np.linspace(1, 100, 1000) // np.linspace(a,b,N) returns 1-d array of N elements with a[i]
= a+ i(b-a)/N
ypts = np.exp(xpts)
plt.plot(x,y)
xpts = np.linspace(1, 100, 1000)
ypts = np.log(xpts)
plt.plot(x,y)
plt.show()
```

3.2 Labels and title

To give the axis different labels simply call the xlabel and ylabel functions found in plot:

```
plt.xlabel(’Time’) // x-axis will have label Time
```

The title function also allows to specify the title.

It is also possible to format the labels and titles in numerous ways.

3.3 Multiple plots

The subplot function allows to plot multiple plots within one figure:

plt.subplot(x,y,z) // constructs figure with x rows, y columns with the current plot the z'th plot in the figure

This function is then called before each call of plot(); in order to caption each plot one simply calls the title function for each plot. To give a title to the entire plot, one calls the suptitle function.