Python 3 Tutorial 1: Using the Command Line

Anthony DeStefano

May 30, 2017

1 Numpy and using the interactive terminal

For a reference on Numpy, see `https://docs.scipy.org/doc/numpy/reference/routines.html`. For more information on Python, see `https://www.tutorialspoint.com/python/index.htm`.

1.1 Scalar variables

To start the Python 3 interactive command prompt, open a new terminal and type python3.

At the Python3 command prompt, type
  a = 3
and press enter. Next type
  b = 5.0
and press enter. To display the value of the variable a, type
  print(a)
To display the variables' values and the product of a and b type
  print(a, b, a*b)

Python is dynamically typed, so the types are inferred internally, unlike C/C++ or FORTRAN. To see what variables are defined, type dir() and globals() separately to see the distinctions. There will be many hidden predefined variables (you can just ignore these). To see the specific variable typecast, type type(a) or type(b), etc.

Typical arithmetic operations are + addition, - subtraction, * multiplication, / division, % modulus, ** exponent, and // floor division. Try testing each of these out with a and b together.

1.2 Array variables

Lists: Lists are built-in Python arrays that can have multiple data types for the variables of each element. An example is
  list_a = ['a','b','c',1,2,3]
where[1] list_a[0] = 'a'. We can also access a subset of an array or list by saying list_a[3:] = [1,2,3] or list_a[:3] = ['a','b','c'], similar to Matlab.

The length of a list is given by the function len(). The + operator acts as a concatenation where as the * operator is the duplication operator for lists, *not* scalar multiplication. Try these two operators on your defined lists. Do the results make sense?

[1] Arrays in Python start at element 0 like in C/C++.

Numpy arrays: Since we will be writing scientific applications, we will use **numpy arrays** instead of the built in list type. The numpy arrays act like vectors, matrices, and Nd arrays, in general, and follow the rules of scalar-vector, vector-vector, vector-matrix multiplication. The functions for dot(), inner(), and cross() products are defined as well as useful transformation functions such as transpose() and reshape(). Note that the reshape() function is very useful for flattening a multidimensional array into a 1D array, as an example.

In order to use the **numpy module** you can type either
```
import numpy as np
```
so that all numpy functions can be accessed using the prefix np. e.g., np.dot(a,b), or
```
from numpy import *
```
so that all numpy functions can be used without the prefix e.g., dot(a,b). The former is useful when there are clashing function names from other modules that are imported[2].

Vectors: A vector of numbers can be defined explicitly by typing
```
x = np.array([0, 1, 2, 3, 4, 5])
y = np.array([1, 2, 3, 4, 5, 6])
```
where x and y are both row vectors. Since the * operator for numpy arrays acts as the element-wise-broadcast operator, if we type
```
z = x*y
```
we will have z = [0, 2, 6, 12, 20, 30]. However, if we want the inner product (without the complex conjugate) between the 1D vectors x and y we can type
```
x.dot(y) or np.dot(x,y)
```
which will return the value 70.

1.3 Matrix variables

Matrices: We can define two matrices by typing
```
v = np.array([[1, 2, 3], [7, 11, 13]])
w = np.array([[2, 3, 5], [-1, -2, -3]])
```
and we can get the size of each dimension by typing
```
v.shape or np.shape(v)
```
which will return (2, 3), meaning we have 2 rows and 3 columns.

Matrix-matrix multiplication: Again, if we attempt to "multiply" the v and w matrices together then the answer will produce element-wise multiplication and *not* matrix multiplication. If we want to do a matrix-matrix multiplication we should use the np.dot() function.

Try v.dot(w). Does this operation work? Why or why not? Now try v.dot(w.T). Note w.T returns the transpose of the matrix w. Does this operation work? Why or why not?

Matrix-vector multiplication: Now define a new vector
```
u = np.array([7, 8, 9])
```
and multiply the vector u with the matrix v by typing both
```
v*u
```
and
```
v.dot(u).
```
In the first case we see that the vector v was element-wise multiplied to both rows of matrix u, while in the second case the **matrix-vector** operation was performed.
It is important to note, if we want a column vector we need to define u as
```
u = np.array([[7, 8, 9]]).T
```
which technically is a matrix of 3 rows and 1 column.

[2]In the tutorials we will use the short-hand prefix notation so that it is clear where each function comes from.

2 Plotting using Matplotlib

For a list of example code using matplotlib, see `https://matplotlib.org/gallery.html`.

The next Python module we will introduce is Matplotlib. Since we will be interested in plotting, we will only import the pyplot package in the matplotlib module by typing

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

The most typical plot is the **x-y plot**. Let us define a range of x-values by typing

```
x = np.arange(0, 10, 0.01)
```

The first argument is the starting point, second is the ending point (exclusive), and lastly the spacing. Now let us define some y-values,

```
y = np.sin(x)
```

and we can produce a plot by calling

```
plt.plot(x,y)
```

Notice that the plot does not display on the screen just yet. We must use the plt.show() function in order to display the plot to the screen. Before we do this, let's set the x, y-axis, and title by typing

```
plt.xlabel('distance [m]')
plt.ylabel('Amplitude')
plt.title('y = sin(x)')
```

If we want to save the figure as an image, we can use the function

```
plt.savefig('sine.png')
```

The file type can be any common image type, such as png, jpg, ps, etc. And finally, we can display the plot by typing

```
plt.show()
```

3 Task

Now that you have completed your tutorial, let's do some science!

Inside your home directory, you should find a file called RecentIndices.txt. Use the Linux terminal to see what is inside the file and what every column of the data is. The first column is the year and the second is the month of the observation. The third column is the observed sunspot number, the 8th is the observed radio flux, and the 10th is the observed AP index (a geomagnetic index). It is suggested the numpy function loadtxt() is used to read the data file.

To complete this task, you must:

1. Generate a png image that contains the observed sunspot number as a function of time, the observed radio flux as a function of time, and the observed AP index as a function of time. Make each set of data as a subplot in a single plot with correct labels for all y-axes and the x-axis with a title. Experiment with changing the colors of the lines.

2. Generate two postscript files (ps), one that contains a scatter plot of the observed sunspot number on the x-axis and the observed radio flus on the y-axis and one that contains the observed sunspot number vs. AP index. Calculate the **correlation coefficient** between the two set of values and include that information in the title of the plot.

Be sure to include correct labeling for all your plots. When you are finished with this task, email the results (3 files) to jacob.heerikhuisen@uah.edu.

Next tutorial preview: In the next tutorial we will focus on defining our own functions and importing these user-defined functions in a Python script. We will also cover how to run this script as a program instead of using the Python interactive terminal.