Introduction to Python and Pylab

Jack Gallimore
Bucknell University
Department of Physics and Astronomy

Table of Contents

Using Python on Bucknell Windows Computers

Much of your work will be in an ipython terminal and a text editor such as Notepad++. To open ipython, use the start menu in the usual way and navigate to **ipython2**. Selecting it will start a new command terminal running the ipython environment. To load the pylab libraries (Numpy, Scipy, and Matplotlib in interactive plotting mode), type the following command in the ipython terminal.

```
%pylab
```

You'll also want to navigate to your directory on Netspace.

```
cd 'U:\\Private\\Python301'
```

Your directory might be different, of course.

Next, use the start menu to open your favorite text editor. Under Windows, I prefer **Notepad++**. However, if you want to use Notepad++, there's a little extra setup to get it to play well with Python scripts.

- In Notepad++, open the Settings menu.
- Under Settings, navigate to Preferences → Tab Settings.
- Make sure 'Replace by Space' is checked.

Using Python on Bucknell Linux Computers

Much of your work will be in a linux terminal and a text editor such as **emacs** or **gedit**. To open a terminal, right click on the desktop and select **Open a Terminal**. This opens a Linux command window in your home directory. Welcome to the world of the command line. Learn this, and your employability increases an order of magnitude.

Once in the command shell, you'll need to run a script to access the latest version of python. Use:

```
source /usr/remote/python/setup.sh
```

or

```
source /usr/remote/python/setup.csh
```

depending on your shell. If you get an error with one command, try the other. (Notice that I am using red for Linux/Unix commands.)

Change your directory to where you want to work. The following example makes a subdirectory called Python301 and changes the working directory to that subdirectory.

```
mkdir Python301
cd Python301
```

You don't have to call your directory Python301, of course, it's just an example.

Start up a text editor in that directory; use one of

```
emacs &
gedit &
```

The “&” symbol puts the program into the background so that you can still use the terminal. You'll need one of those editors to edit Python scripts in your working directory.

If you downloaded any scripts from the Moodle page, they will likely be in your home directory. You'll need to copy them to your working directory. For example,

```
cp ~/constants.py .
```

will copy the file constants.py from your home directory (~) to the current directory (.).

To start ipython with pylab libraries loaded, use the following command.

```
ipython --pylab
```

Now you are in an interactive Python environment that behaves like the Linux command prompt but instead employs Python commands. You can enter Python commands interactively as we discussed in class, or you can run a script using the `execfile` function. For example,

```
execfile('myScript.py')
```

runs the script called myScript.py. (Notice that, in this document, Python commands use the same font as Linux/Unix commands, but are colored black.)

If at any point you want to review a history of what you have done in ipython, use the %hist command as follows.

```
%hist -n
```

To get out of ipython, use **ctrl-D**.

Writing Python Scripts

Interactive Python vs. Scripting

Interactive Python (ipython)

Interactive Python is useful primarily to perform quick calculations. For example, you could import the constants.py module and perform numerical calculations using symbols for physical constants.

Interactive Python is also useful for debugging scripts. If a script crashes, you will be returned to a Python command prompt and all *global* variables retain their values. You can print their values interactively to try and hunt down the problem.

Scripts

If you want to perform the same calculation many times, or a more detailed calculation requiring many steps, you are better off writing a script that you can edit and run at will. Such tasks require repetition, or loops, and a script more easily handles loops than an interactive session.

A script is just a (unicode-based) **text-file containing a list of python commands that are executed in sequence.** Any command that you can execute in ipython can also be executed within a script. The main advantage of a script is that you have to type the commands only once, and then they can be repeated as often as you are willing to run the shorter command execfile.

It is possible in Python to write an object-oriented script that permits asynchronous (non-sequential) operations, such as programming GUI interfaces. We won't work with objects and GUIs in ASTR/PHYS 301.

There's nothing all that special about writing a script. If you put a list of python commands into a text-

file, you can execute it using the execfile command. From within ipython,

```
execfile('myScript.py')
```

and the commands inside the file 'myScript.py' will be executed in sequence.

There is no strict requirement for the name of a python script, but it is good form and will save headaches later on to give the script a ".py" extension.

You can also execute a Python script from a system command prompt by using the "python" system command.

```
python myScript.py
```

The disadvantage of using this method, especially when you are developing a script, is that all of the variable data are lost if the script crashes, making it tougher to debug the script. The advantage of running scripts from within ipython is that the variable data are retained in the event of a crash.

Python Programming Basics

Variables

In Python, variables are more commonly dynamically assigned. Formally, a variable is just a set of locations in computer memory that stores a value or a string of characters. That variable is a shorthand symbol assigned by the programmer, like the variables *x* and *y* from algebra.

By dynamic assignment, I mean that you don't have to allocate memory for a variable at the beginning of the program. You can define a variable anywhere in your script, and Python will automatically allocate memory for it.

Assignment Statements

In all programming languages, the "=" symbol does not mean the same as in mathematics. The "=" symbol implies assignment: whatever lies on the RHS of the "=" is stored in the variable on the LHS. Assignment statements will sometimes read like nonsense if you try to interpret them mathematically. Examples:

```
x = 3
```

This statement makes sense: we are defining a variable x and assigning the value 3 to it.

```
x = x + 1
```

Mathematically, this statement is nonsense, because it implies that 1 = 0. In a programming language, the interpretation is different: first evaluate the RHS (x+1 = 4) and assign it to the variable on the LHS; the result is x = 4.

Python can also perform multiple assignments simultaneously.

```
x, y = 3, 5
```

is the same as the two commands

```
x = 3
y = 5
```

Floating-Point vs Integer Arithmetic

Be careful when you assign variables and perform calculations. Try the following examples.

```
x = 2
y = 3
print x / y

x = 2.0
y = 3.0
print x / y
```

If you want decimal (floating-point) division, you should enter your numbers with a decimal point. Otherwise, the calculations will be truncated or floored to an integer.

Strings

You can also assign strings to variables, such as

```
myString = 'This is my string.'
print myString
```

Some special characters won't work inside a string without an “escape.” For example, if you want to put an apostrophe into a string, you have to escape it as follows,

```
myString = 'You haven\'t seen my string, have you?'
```

The '\' symbol tells python to interpret the following character as something to be stored as a string rather than a Python character. (Here, if you didn't escape the apostrophe, it would have closed the string after 'n.')

One use of strings is to format output statements, such as

```
print 'The value of variable x = ', x
```

Lists and Arrays

In Python, there are many ways to store sets of numbers in a variable; for now, we'll concern ourselves with lists and arrays.

Lists are just lists of values stored in a single variable. For example,

```
a = [0, 1, 2, 10, 25]
```

or

```
b = ['one', 'two', 'three', 'potato']
```

If you want to add new elements to a list, use + and another list.

```
b = b + ['rutabaga', 'marshmallow']
print b
```

The range function allows you to define a set of sequential integers.

```
a = range(10)
```

This command is equivalent to

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Notice that the argument is the length of the array, not the highest value.

To access an individual element of an array, you have to use an index, which is the position of the element that you want. For example, a[0] accesses the first element of the array a, a[1] the second, and so on.

Arrays are unique to numpy and offer the advantage of quick calculations. To define an array, you use the array function around a list variable.

```
a = [0, 1, 2, 3] # list
b = array(a) # array containing the same values
```

Suppose I wanted to square each element of `a` and store the values in a new list, `c`. Since `a` is a list, I have to perform a loop (see below), and square each term in sequence. With array b, however, it's easy.

```
c = b**2
print c
```

With numpy arrays, calculations are automatically performed for each element in sequence without having to use a loop.

Functions

In computer programming, “function” has a different meaning than in mathematics. A function is just a virtual machine that takes a variable or set of variables, performs some operations or calculations, and returns some new value or set of values. For example, here is a function that calculates the hypotenuse of a right triangle, given the sides `a` and `b`.

```
def hypotenuse(a, b):
     r = sqrt(a*a + b*b)
     return r
```

Notice how it's defined: the def statement tells python that a function statement is going to follow. Next is the name of the function, followed by a list of arguments that the function needs to perform the calculation. The colon, “:,” tells Python that the commands of the function follow.

The commands within a function are indented – indentations in Python indicate commands that belong to the function (or, as we'll see later, loops).

As long as this function is defined in the script before the main program, we can then call it in the main program as follows.

```
print hypotenuse(3., 5.)

q = 2.0
w = 10.0
print hypotenuse(q, w)
# notice that functions can take appropriate
# variables as arguments
```

Functions can of course be much more complicated, and an individual function can be longer than the main program that calls it.

Loops

Loops are programming structures that allow you to perform repeated calculations. For our purposes, we will concern ourselves with two kinds of loops; for loops and while loops.

For Loops

For loops iterate over a restricted, well-defined set of values. In Python, the method is to have some predefined list of values that you want to perform calculations or operations on.

Go back to the list example, from above.

```
a = [0,1,2,3]
```

Suppose you wanted to square each element and store the result in a new list `c`. Notice how commands within the loop are indented, just like in functions.

```
c = [] # declare an empty list
# this for-loop takes each element of a in turn
# and stores it in the variable value
for value in a:
     c = c + [value**2] # add the value squared to the list
print c
```

Hopefully now you can see the advantage to numpy arrays: with an array, you don't need to use a for loop for simple math operations.

Sometimes you'll want to loop over the value of the index of an array, rather than the array value itself. For example, a and c are now two lists with the same length. If we want to print each element of each array side-by-side, we'll want to access each element by its index rather than its value.

In this case, we use the len command to define a set of indices to iterate over.

```
n = len(a)
for j in range(n):
     print a[j], c[j]
```

While Loops

While loops are more useful when you are repeating operations over an set of values that are not predefined or may depend on calculations within the loop.

I'll use Zeno's paradox for this example. In Zeno's paradox, you imagine a frog hopping toward a pond. The frog is subject to peculiar jumping arithmetic: for each hop, it can only hop half of the remaining distance to the pond.

Suppose the pond is 10 meters away, and the frog itself is 0.05 meters long. Our while loop will consider how many hops it takes the frog before it is less than its length from the pond: at that point, we can consider that the frog has reached the pond.

```
nhops = 0 # initialize our counters
distance = 10.0
length = 0.05
while(distance > length):
     distance = distance / 2 # hop half the remaining distance
     nhops = nhops + 1       # update the hops counter
     print nhops, distance
```

You should try this one yourself. I get eight hops before the frog is within its length of the pond.

Notice that the argument for a while loop is *boolean*: it is a logical statement that can be evaluated as true or false. Boolean statements are easy in python: the one catch is that if you want to establish equivalence, you have to use "==" instead of "=". A while loop terminates when the argument is False.

Boolean Statements

Here are some Boolean (logical) statements. You should try typing them into an ipython session to verify that the results make sense.

```
1 == 2
1 == 1
1 > 2
1 < 2
1 <= 2
(1 <=2) and (2 <= 3)
(1 <=2) and (2 > 3)
(1 <=2) or (2 > 3)
```

Conditional Operations

Your program may have to make some choices as it runs. The if-statement sets rules for these choices. For example, suppose you want to square *x* if it is less than 1, but cube it if it is greater than 1.

```
y = x**2
if x > 1:
     y = x**3
```

Alternatively, you could use the else-statement.

```
if x>1:
     y = x**3
else:
     y = x**2
```

You can include even more conditional options using the elif-statement.

Notice that an if-statement follows the same structure as a while loop, including a boolean argument, except that it is not executed repeatedly but conditionally.

Structure of a Python Script

For the purposes of this course, all python scripts should follow a strict outline, namely, (1) module imports, (2) function definitions, and (3) the main program. The following script illustrates the outline.

```
#!/usr/bin/env python

import constants as c
from pylab import *
# other import statements

# function definitions, for example

def xSquared(x):
     y = x*x
     return y

def xCubed(x):
     return x**3

def xToTheThreeHalves(x):
     return pow(x, 1.5)

# the main program
yarray = arange(5)
for y in yarray: # for loop
     print y, xCubed(y), xToTheThreeHalves(y)
     if y > 3:
          print 'Hey, y is greater than 3!'
print 'All done!'
```

Variable Scope

Variables and functions will have some *scope* within your script. Variables with *global* scope can be accessed anywhere in the program. Variables with *local* scope can only be accessed from within the function where they are defined.

Some basic rules:
(1) Any modules or variables that are imported are *global*. They can be used anywhere in the script that follows. That's why it is a good idea to have your import statements up front.

(2) Any function definitions within a script are *global*: they can be accessed from anywhere in the main program that follows. That's why it is a good idea to have your function definitions *before* the main

program, to ensure you can use your functions. However, you want your *imports* before your functions, so you can use the imports in your functions.

(3) Variables within a function are defined locally. For example, looking at the example script above, the function xSquared has the argument *x*, and an internal variable *y*. These variables are *local* to the function: they are assigned when you call the function, and they occupy accessible memory only for the brief time that the function is performing the calculation. Local variables in a function are called *volatile*, because they are lost once the function is done with the calculation. However, the return statement sends local variable values back to the main program, where they can be accessed.

Example: Trapezoid Integration

Trapezoid integration is a practical application of what we have covered so far. In trapezoid integration, we replace a function with a set of trapezoids and perform Euler step integration.

The following code integrates the function x^2 from 0 to 1 using trapezoidal integration. We know in advance the answer should be 1/3. (If you want to integrate a different function, just alter the function `myFunc`).

```
#!/usr/bin/env python

import constants as c
from pylab import *

def myFunc(x): # function to be integrated
     return x**2

# define integration interval
a = 0.0
b = 1.0

# define step-size
dx = 0.1

# Note: the area of a trapezoid = 0.5 * base * sum of heights
x0 = a
integral = 0.0  # initialize integral
while(x0 < b):
     x1 = x0 + dx
     y0 = myFunc(x0)
     if x1 > b: # don't go past the end of the interval
          x1 = b
     y1 = myFunc(x1)
     area = 0.5 * dx * (y0 + y1)
     integral = integral + area
     x0 = x0 + dx # slide trapezoid to the next position
```

```
print integral
```

In-class exercises: The Factorial Function

(1) Make a factorial function and test it in ipython. Difficulty: don't use the built-in factorial functions.
(2) Modify the factorial function so that it returns the natural logarithm instead – the built-in natural log function is log().
(3) Modify the factorial function so that, for arguments above $n = 20$, it uses Stirling's approximation to calculate the log factorial.

Stirling's approximation

$$\ln n! \approx 0.5 \ln(2\pi n) + n \ln(n) - n.$$

File I/O

In python calculations, you are likely to generate large tables that you would like to be able to store and inspect later. In other words, we need to save your calculations to a file. In the following discussion, I'll concentrate on data that are stored in columns. For example, if you are calculating the 1-D motion of an object, you would like to have columns for *t* and *x*.

```
# t    x
0.0   0.0
0.1   0.2
0.2   0.5
0.3   0.633
.
.
.
```

Writing columnar data to a file involves three steps:

(1) Opening the file. Essentially, we pick a name for the file, and assign it to a special variable called a file pointer, which I'll call pointer for short.

(2) Printing data to the file, usually within a loop.

(3) Closing the file. As data are printed to a file, they are temporarily stored in a reserved area of memory called the cache. By closing the file, we write the remaining data in the cache to the file on disk, and we also free up the file pointer variable for later use.

For example, suppose we have data in the arrays `t`, `x`, `y`, and `z`. The following code snippet writes their data as neighboring columns in a text file.

```
# step 1: assign the file pointer fp to 'data.txt',
# open for 'w'riting
fp = open('data.txt', 'w')
for i in range(len(x)): # loop over all indices of the array
```

```
    print >>fp, t[i], x[i], y[i], z[i] # step 2
fp.close() # step(3)
```

Notice that we use the familiar command `print`. The symbol "`>>`" tells `print` to redirect the output to the file.

With Numpy, reading those same data back into python is easy: use the `loadtxt` command.

```
data = loadtxt('data.txt') # loadtxt automates input
# data is now a matrix containing all of the columns.
# Copy those columns to arrays
t = data[:,0] # first columns
x = data[:,1] # second column
y = data[:,2]  # third column
z = data[:,3] # fourth column
```

There are ways to read the data in one line at a time, which is useful for more complex data formats (data that includes a mix of strings and floating point numbers, for example). For our purposes, we are better off sticking with columnar data and reading them in using the `loadtxt` command.

Numpy Shortcut for File Output

Numpy's `loadtxt` command reads columnar data from a text file, and `savetxt` is a shortcut for saving data to a text file. There's a catch, however, if your data are in separate variables instead of a single array. We have to use the `zip` command to merge vectors into an array. Here's an example in which I define a set of vectors `t`, `x`, and `y`, and then I use `savetxt` + `zip` to save the data to a columnar text file in one line.

```
t = arange(0,100) / 10.
x = sqrt(t)
y = t**2
savetxt('myDataFile.txt', zip(t, x, y))
```

Euler Step Integration

Back in PHYS 211, you had to program an Excel spreadsheet to integrate the 1-D equations of motion for a falling object subject to air drag. The force law in this 1-D case is,

$$F_x = mg - bv_x^2,$$

where we define downwards to be the positive *x* direction and *b* is the drag coefficient.

To numerically integrate the force law and derive the resulting motion, we start with Newton's second law to get the acceleration, and then we carry out calculations over very small time-steps *dt* such that we can assume that the accelerations is roughly constant during that time step. In this limit, the equations of motion become,

$$
\begin{aligned}
a_x(t) &= F_x(v_x)/m, \\
v_x(t+dt) &= v_x(t) + a_x(t)\,dt, \\
x(t+dt) &= x(t) + v_x(t)\,dt,
\end{aligned}
$$

where the explicit functional dependence is denoted by parentheses; i.e., $x(t+dt)$ indicates "x at time $t + dt$," **not** "x times $t+dt$." Note that these equations could be applied to any force law, and furthermore they can be extended to 2-D and 3-D force laws (i.e., force laws that include y and z components).

The following Python script performs Euler step integration for the 1-D air drag problem. The data are stored in lists, written to a file, and plotted. This provides a nice summary of everything we've covered so far.

```
#!/usr/bin/env python


# imports
from pylab import *


# function definitions
def forceLaw(v, m, b):
    g = 9.8 # m/s**2
    F = m*g - b*v*v
    return F


# main program


# Describe the object
b = 0.005 # kg / m
m = 0.008 # 8 grams


# initialize the Euler steps (boundary conditions)
t = 0.0
dt = 0.1 # seconds. Could be smaller.
x = 0.0
v = 0.0 # drop from rest
a = forceLaw(v, m, b) / m  # initial acceleration


# define lists to store data
```

```
tList = [t]
xList = [x]
vList = [v]
aList = [a]

# integrate until terminal velocity is reached
# Define terminal velocity acceleration < 1.e-6
while(a >= 1.e-6):
    # Euler steps
    x = x + v * dt
    v = v + a * dt
    t = t + dt
    # Update (append) the lists
    tList = tList + [t]
    xList = xList + [x]
    vList = vList + [v]
    aList = aList + [a]
    # update the force law and acceleration to use for next step
    F = forceLaw(v, m, b)
    a = F / m

# write the data to a file
fp = open('eulerExperiment.txt', 'w')
for i in range(len(tList)):
    print >>fp, tList[i], xList[i], vList[i], aList[i]
fp.close()

# plot the results
# first, position plot
clf()
plot(tList, xList)
xlabel('Time (s)')
```

```
ylabel('Position (m)')
savefig('posPlot.pdf')

# velocity plot
clf()
plot(tList, vList)
xlabel('Time (s)')
# use TeX formatting on the string
ylabel('Velocity (m s$^{-1}$)')
savefig('velPlot.pdf')

# first, position plot
clf()
plot(tList, aList)
xlabel('Time (s)')
# use TeX formatting on the string
ylabel('Acceleration (m s$^{-2}$)')
savefig('accPlot.pdf')
```