Getting Started with Python!

Developed 05/21/2018 by B. DePaola

Installing Anaconda

The first thing you need to do is to install Anaconda. While not technically required for to run Python, Anaconda is a convenient starting point because once you've installed Anaconda, you've installed Python itself, Jupyter, Spyder, and more or less all of the Python modules that you will need for scientific computing. More on these other packages later. For now, let's install!

The first decision you'll need to make is whether to go with Python 2.x or Python 3.x. Normally I'd automatically recommend going with the newest package of any piece of software. In the case of Python, however, many scientist have developed very useful packages under Python 2.x, which means that many other scientists refuse to move to Python 3.x -- hence propagating the dual-version problem. (I personally fault the Python developers for not doing a better job in making Python 3.x *backward compatible*.) The bottom line is that you will need to be able to move between the two different versions. Furthermore -- and this is perhaps the most important consideration -- the Python developers plan to completely stop maintaining Pythong 2.x after 2019. Python 2.x will no longer receive security updates nor will it receive new features. Furthermore, we are told that open-source maintainers are dropping Python 2.x support for their libraries.(See https://tdhopper.com/2to3 (https://tdhopper.com/2to3)). But never fear: when we discuss a function or syntax that the two versions treat differently, we'll try to alert you. For now, pick a version. If you choose 2.x you'll want to install Anaconda2; if you choose 3.x you'll want to install Anaconda3. Note that if you install 2.x, there is a work-around that lets you access 3.x features. The reverse is not true: There is no work-around that gives 2.x compatibility to 3.x. **I personally recommend 3.x.**

First point your browser to https://www.anaconda.com/download/ (https://www.anaconda.com/download/). Choose the Linux, Windows or macOS, choose Anaconda2.x or Anaconda3.x and hit the install button. Then follow the directions. Easy peasy.

If you find at some point that Anaconda does not have some Python package that you'd like to have, the easiest way to install that package is by using the conda package handler from the command prompt. (In Windows, you can open the commmand window by typing cmd.exe from the start menu; in the macOS you open up Terminal.app; from Linux you open a "terminal" window. In all cases, once you have a command prompt, you simply type

conda *package*

where *package* is the name of the package you'd like to install. For example, installing Anaconda automatically gives you the powerful graphics package called Matplotlib. But if it didn't, you could load it in by typing

conda matplotlib

from the command line. It's that simple.

Once Anaconda has been installed, we can use the Anaconda Navigator to launch Jupyter, or Spyder. First let's briefly discuss Jupyter.

Jupyter

Jupyter is a so-called notebook environment. A notebook is a way to combine operational code with beautiful documentation. A Jupyter notebook is broken up into *cells*. A cell can contain either documentation, written in the *markdown* language, or executable code. The code can be in one of several different programming languages, but we'll concentrate on Python. Even though the notebooks are local (they're stored on your local computer, not on the web) Jupyter uses your default browser as its software platform.

The markdown language is sort of a hybrid of HTML and LaTeX. Youv'e probably heard of the former; the latter is a powerful way of depicting mathematical expressions. For example, suppose we are discussing integration and we'd like to the integral under discussion. We can easily express it in LaTeX:

$$y = \int_{-\infty}^{\infty} \frac{dx}{\pi^2(x^2+1)(a-x)^2+1}$$

You can readily switch back and forth between edit mode and display mode. For example, to switch this display to edit mode, put your cursor in this cell and hit *enter* twice. Do this and look at the above equation. If it looks scary, don't worry; LaTeX, while extremely powerful, is much easier to generate than it seems.

To get back to display mode, either click on the right-arrow symbol above or (even quicker) hit *enter* while holding down the *shift* key.

The tutorials in this class will almost all be done via Jupyter notebooks. The real advantage of which are not just the documentation mode, but also the executable cells. Look at the following cells:

In [ ]:
```
a = 5
a
```

In [ ]:
```
b = 3*a**2
b
```

You can execute the code in a cell by clicking on the cell of interest and then either hitting the right-arrow icon or *shift-enter*. Execute each of the above two cells.

We'll get back to Python very soon, but first

Spyder

Spyder is an example of an IDE (integrated development environment). When you open up Spyder, you will probably get three panes: A *script* pane on the left, an *IPython* pane on the lower right. (Your implementation may vary from this but you can move the panes around as you wish.)

First let's discuss the IPython pane. IPython stands for "interactive Python". It's a nice place to test built-in functions and expressions to see if they work the way you expect. It can also be used as a sort of "calculator". Basically, you just type something at the prompt and Python responds.

While IPython is very convenient for testing and for one-liner computations, most of your applications of Python will involve writing a "program" (technically, in Python this is called a "script".) For this you need some sort of basic text editor. Like most programming languages, Python commands are written in "plain text", that is, unlike what Word or other word-processing packages do to what you type, editors used for generating programs or scripts do not embed so-called control characters within what what you type. For example, if you were to somehow dump a Word document, such that you could examine it character by character, you'd find that many of those characters can't even be printed; the so-called ASCII code doesn't correspond to any characters on your computer's keyboard. Word does this in order to embed commands that are useful in word processing, indicators for font types, italics and bold face, indentation, etc. Python doesn't need (or want) all of this embedded in its scripts, so you need an editor that doesn't insert anything without your knowledge. Spyder's editor only generates plain text.

But it does far more! Spyder's editor color codes commands to help you find bugs and to help you see useful patterns in your scripts. For example, here's a summary of what colors mean what in Spyder's editor:

Color	Meaning
Green	Comment block
Gray italic	Comment line
Blue	Key word
Brown	Constant objects
Black	Everything else

Spyder's editor has a great many features. It allows you to define and execute individual blocks of scripts, and has some powerful debugging features as well. We'll get into these as we work our way though the Python language.

In general, I use Spyder for developing scripts and Jupyter when my scripts need more documentation than casual commenting allows.

Importing Modules

Since its creation in 19xx, many users have contributed powerful add-ons to Python's core commands. These are available in *modules* that you can *import* as needed. There are several ways to import modules -- and you need to know them all if you want the maximum flexibility that Python allows. The following table summarizes the import syntaxes along with what they do. In this table, **numpy**, **scipy**, **mathpy**, **matplotlib**, and **pyplot** are various modules; **sqrt** and **exp** are functions in a module.

Import Syntax	Function
from numpy import *	Imports the entire module named "numpy", including all the submodules it contains. Ex: **sqrt(7)**
import numpy	Like "from numpy import *" but *requires functions to be preceded by "numpy.*" Ex: **numpy.sqrt(7)** takes the square root of 7.
import numpy as np	Same as "import numpy" but requires "np." before the function. Ex: **np.sqrt(7)**
from numpy import sqrt	Imports only sqrt from numpy. Does *not* require "numpy." before the function. Ex: **sqrt(7)**

From past experience, I know that the subtle distinctions between these different import syntaxes cause confusion, so let's go over them one by one. At the same time I'll offer best-practices suggestions as to when to use which.

- **from numpy import *** : Inelegant but occasionally useful. Here's the problem: suppose you use this command *and* **from mathpy import *** . Both of these functions contain the **sqrt** function. But these functions behave differently when dealing with arrays! So how does Python know which one to use? (It doesn't!) Except in very rare circumstances I recommend you do *not* use this syntax!
- **import numpy**: Safest but klunkiest. Every time you use a function from the module, you need to precede that function by the module's name. For the numpy module this is not terrible, but imagine typing out "matplotlib." before every plotting command!
- **import numpy as np**: Gives you the safety of the previous syntax, but the convenience of using your own shorthand instead of typing out the entire name of the module. For example, it is much quicker to type out **plt.plot()** than **matplotlib.plot()**. The only downside is that if you are cutting and pasting a code snippet that someone else wrote, your choice of shorthand may differ from theirs. That's ok if you know this problem can happen, and it's easily fixed by a global "find and replace". This is the way I usually go.
- **from numpy import sqrt**: This is safe and convenient, but only if there are a limited number of functions that you want to import from a module. For example, I use the syntax when I need just one kind of integration routine from a module that contains a large variety of integrators.

Now that we know something about importing modules, let's adopt an importan habit: start each and every Python application with the following imports:

In [3]:
```
# -*- coding: utf-8 -*-
"""
Purpose:
Created on %(date)s
Modified:  %(date)s

@author: %(username)s
"""
from __future__ import division, print_function
input = raw_input

import numpy as np
import matplotlib as plt
```

I recommend that each and every script you write begins with these lines of code (empty lines are optional!). Fortunately, Spyder helps you automate this process. More on that in a moment, but first let me explain lines 1-8: The first line is just weird. Normally in Python, everything to the right of a hash symbol ("#") is a comment and is therefore ignored by Python. This line is an exception. It is a complicated way of telling Python where to find non-English characters and accents if you decide to use them. (For greater portability, I use only standard characters and no accents. But because I can automate this line's appearance, and because it costs me nothing to do so, I usually try to include it.)

The appearance of three consequative double-quotes tells Python that you are beginning or ending a *comment block*. That is, after you've typed """, everything that follows will be ignored by Python until it encounters another """. This is very convenience because it allows you lots of flexibility to include all sorts of information. I consider it imperative to include the information shown above. And if you include the above 10 lines in a sort of template, then you will be reminded to include that information in your scripts.

Now, here's how you can create that template: From the Spyder environment, click on the wrench icon on the top of the page. On the left side of the drop-down menu, select **editor**. Then on the right side select **Advanced settings**. The click on the button **Edit template for new modules**. Then edit what is there to match the following and hit the save button:

```
# -- coding: utf-8 --
"""
Purpose:
Created on %(date)s
Modified: %(date)s

@author: %(username)s
"""

from future import division, print_function
input = raw_input

import numpy as np
import matplotlib as plt
```

The "%(date)s" and "%(username)s" will automatically be filled in by the appropriate dates and by your user name everytime you implement the template. What could be easier?

Following the comment block are two lines that represent a work-around to 3.x if you've installed 2.x. **Do not use these two lines if you've installed 3.x on your machine.** The final two lines import from the extremely valuable Numpy and Matplotlib modules. **I'd recommend you *always* import them** for any sort of scientific computing.

Magic

This is not what you're thinking. In IPython (and in Jupyter, which employs IPython) magic commands can be very useful. These are not programming commands but, rather, are directives to IPython itself. The most commonly used magic command is "%reset" and is used to, essentially, start fresh. The %reset command "undefines" all variables and functions that had been defined. It also "unimports" all modules. It's really like exiting Python and then going back in.

All magic commands begin with a "%". If you want to see the complete list of magic commands, you can type %magic at the IPython prompt. Besides %reset, we won't make much use of magic commands in these Python lessons.

Name-Calling

Again, not what you're thinking. When recently surveyed as to what is the most difficult part of programming, professional programmers overwhelming stated that it was coming up with names. Names for variables, names for programs, names for functions, all these entities must have names -- and best practice is to use names that "self-document" by making clear what variable, program, or function does. Coming up with descriptive names seems especially for inexperienced programmers who are comfortable in algegraic manipulations. After all, we wouldn't think twice about creating a function **f(x)** where **f** is the independent variable and **x** is the dependent variable. Indeed, in algebraic manipulations there is *value* in leaving variable names abstract and therefore more general.

But in programming, usually just the opposite is true. If I have a variable that represents, say, momentum, I should probably *not* use **p** as the variable name. Choosing **momentum** as the variable name will be clearer and less ambiguous. Why is this such a big deal? It all comes down to code documentation. The more judicious your choice of variable names, the less explicit comments you need for the same script clarity. Furthermore, as your scripts grow longer, you may forget which names you used for which variables. But keeping your variable names descriptive helps.

CamelCase versus Snake_case

In our quest for more descriptive names for variables, functions, etc, we can resort to multiword names. There are three main ways that people do this. The first is just to glom together the words. For example oldphi could refer to "a previous value of ϕ". This can be a good way to make your variable names more descriptive, but can be difficult to read. For example, consider the variable name "bobseagle". Does this represent "bob's eagle" or "bob Seagle"? Ok, a silly example, but you get the idea.) One way of combining multiple words is to "camelCase" them. This means starting each new word in the variable name (with the possible exception of the very first letter of the variable) with a capital letter. The preceding example would then be written "bobSeagle" (or bobsEagle, depending on what information I wanted to convey).

Another option is to separate words in a variable name with an underscore, for example bobs_eagle etc. Which you employ is up to you, but it pays to be consistent; otherwise in long scripts you may forget if you referred to your old value of ϕ as oldphi, oldPhi, or old_phi, and you'll waste too much time searching through your code to see what you did.

Ok, we've spent more than enough time on the preliminaries. Let's learn Python!

In [ ]: