Introduction to Python

March 27, 2015

1 Outline

- Introduction to Python
- Installing Python
- Installing IPython
- Installing additional libraries
  - NumPy
  - pandas
  - Matplotlib
- A brief overview of syntax in Python
- Some Labs

2 Installing Python

For this course, we will use `Python 2.7` over `Python 3.4`. Why is this? No particular reason, although `Python 2.7` is still the overall favorite of most developers. We will only be covering tools that are agnostic to the `2.7-3.4` war, so it shouldn't matter much.

If you plan to install Python to your computer locally, you will have to install not only the Python libraries and interpreters themselves but the additional software that we will use throughout the course. This is a huge pain, and while this can be done manually, we highly recommend you use a third-party package management program.

We have experimented with each of the different package management tools, and our favorite is Anaconda, provided by Continuum analytics. However, you can make your set-up work with any package manager. We just think it's the easiest with Anaconda.

2.1 Anaconda

To install Anaconda, go the Continuum Store and select "Download Anaconda." This will require you to enter an email address, though this does not have to be linked to promotional emails. After doing this, you will be sent to a page to download the correct version for your local operating system. Download the correct installer for your OS, and run the corresponding installation programs once it is complete.

For Unix users, this means moving to the Downloads directory and entering

```
bash Anaconda-2.1.0-[Linux or MacOSX]-x86_[32 or 64].sh
```

The instructions are on the download page. For Windows users, just double-click the `.exe` file and follow the requisite instructions.

Once Anaconda is installed, open a terminal and type

```
conda update conda
conda update ipython ipython-notebook ipython-qtconsole
```

This should install the IPython software we will use throughout the course, while also updating all of the Python libraries.

2.1.1 Wakari Cloud

If you don't want to install `Python` packages locally, Continuum Analytics has a cloud-based solution, Wakari. It's not designed for extensive individual use, and it hasn't updated IPython to the latest version, but it is permissible for our course. To get started, make an account on Wakari.io, which will give you your own directory for IPython projects.

2.2 Canopy

Canopy is an alternative to Anaconda, which includes its own integrated development environment on top of a package manager. We don't recommend using Canopy because we found it difficult to update to the latest version of the IPython notebook, and there seems to be issues with the `matplotlib` engine outside of the Canopy IDE.

However, if you desire, you can install Canopy by going to the Enthought Store and selecting "DOWNLOAD Canopy Express". This will require that you make an account with Enthought. One benefit of making an account and then identifying yourself as a student is that it can give you access to tutorial videos on `Python` and its data analysis libraries (this privilege can be finicky, however).

From there, simply install the Canopy program that you download from the site, and open it. Canopy gives a graphical package manager, which you might find appealing (we don't).

2.3 Manual installation

Manual installation is, simply put, a pain in the neck. We don't recommend doing this, because it is quite complicated. To be fair, we will call using `pip` as manual installation, even though this simplifies true manual installation quite significantly. In fact, we won't go into details (to discourage you from trying). It can be done, by following (roughly) these steps:

- Downloading and installing `Python 2.7` (or `3.4`) from the Official Website
- Hopefully, after the installation, you can call the program `python` from the terminal. This is not guaranteed. If you are lucky, download get-pip.py, and run `python get-pip.py` from the relevant directory.
- This should install `pip`. From here, run

  ```
  pip install numpy
  pip install scipy
  pip install matplotlib
  pip install pandas
  pip install ipython[all]
  ```

- If the gods are smiling on you, this will install everything you need for this class, and you should be good to go.

2.4 A simple `Python` script

Open a terminal (Windows users: your "terminal" is technically called the Command Prompt, `cmd.exe`, but we will use "terminal" colloquially) and enter `ipython`. Check to see that the packages we will be using in this course are properly installed by entering the following code into the terminal:

```
ipython
```

If everything is installed properly, this should launch the `IPython` shell. This is a simple interpreter, which is handy to use from time to time, but we will not use it in this course. However, we can take this opportunity to write a simple line of code into the interpreter:

```
print "Hello, world!"
```

Congratulations, you have officially been inducted into the wonderful world of computing in `Python`!

3 Python basics

Python works like other programming languages in that it has some built-in types:

- int, an integer type (default precision: 32-bit)
- float, a floating-point type (default precision: 64-bit)
- long, an integer type (unlimited precision)
- str, a string literal type
- Boolean, a boolean type

Variable initialization in Python is different than the C family of languages because variable type is **not** specified. In C, you have to declare the type immediately:

```
double var = 1.61;
```

The equivalent Python code is

```
var = 1.61
```

3.0.1 Try it!

Open an IPython Notebook file.

```
ipython notebook
```

Use the next few examples to get a feel for Python's style of variables.

```
In [6]: a = 10
        b = 20
        print a + b
        print a == b
        print not ( a > b)
```

```
30
False
True
```

3.1 Arithmetic

Python arithmetic is what you would expect. +, -, *, / are all defined as normal, but the operation x ** y is equivalent to x^y.

3.2 Pointers in Python:

What do you expect will happen when you run the following code?

```
In [7]: x = 10.42
        y = x
        x += 12.
        print y

        str1 = "Hello"
        str2 = str1
        str1 += " world!"
        print str2
```

```
10.42
Hello
```

In `Python`, everything is a pointer, but `Python` will try to behave like you intend it to (no horrible `C`-like issues).

Boolean expressions in `Python` are designed with readability in mind. What do you expect the following output will be?

```
In [8]: x = 1.23
        y = -0.5
        bool_one = (x >= y)
        bool_two = (y ** 2 + 2 < x)

        print (not bool_one) or bool_two
        print bool_one is not bool_two
```

```
False
True
```

3.3 Lists

There is not a native array type in `Python` like there is in `C` and `Java`. Instead, `Python` follows a similar model to `Scheme` in that the core native "iterable" data type is a (linked) list. Lists are initialized in `Python` by

```
my_list = [1, 2, 3, 4, 5]
```

Lists indices begin at 0 (Sorry, `MATLAB` and `R` users), so `my_list[0] = 1`. Lists are endowed with some very useful functions:

- `append`, which adds an element to the end of the list.
- `index`, which searches a list for a particular value.
- `insert`, which adds an element after a specified location.

One important concept in `Python` is list *slicing*, which allows you to quickly access sections of a list without much code overhead. For example, if I wish to obtain the middle three elements of `my_list`, I can simply type

```
mid_three = my_list[1:4:1]
```

The braces `[1:4:1]` should be read as telling the list to start at the 1th (that is, second) index and go until the 4th (that is, fifth) index, incrementing by 1. So, the 1 is "inclusive", whereas the 4 is "exclusive." One can reach the last element of a list easily without knowing the length of the list by writing `my_list[-1]`. In general, the `-` indicates that you want to traverse the list in the opposite direction.

3.3.1 Try it!

What do you expect the following code will produce?

```
In [9]: example = [3., 5., 7., 9., 11., 14.]

        example.append(82.)

        example.insert(3, 1.)

        print "Example: \t", example
        print "\t\t", example[0:2]
        print "\t\t", example[3:]
        print "\t\t", example[::-1]
```

```
Example:         [3.0, 5.0, 7.0, 1.0, 9.0, 11.0, 14.0, 82.0]
                [3.0, 5.0]
                [1.0, 9.0, 11.0, 14.0, 82.0]
                [82.0, 14.0, 11.0, 9.0, 1.0, 7.0, 5.0, 3.0]
```

3.4 Conditionals

`Python` uses `if-elif-else` branching very much like the `C` family uses `if-else if-else` statements. There is no `Python` equivalent to `switch`.

```
if True:
    print "this will happen"
else:
    print "this will not happen"
```

Notice that `Python` is sensitive to white-spaces. The `IPython Notebook` is clever enough to handle proper spacing for you.

3.4.1 Try it!

What do you think will happen when you execute the following snippets of code?

```
In [10]: if 1 < 0:
             print "1 < 0"
         elif 1 > 2:
             print "1 > 2"
         else:
             if 1 <= 1:
             print "1 <= 1"


           File "<ipython-input-10-d824fa9aecd4>", line 7
         print "1 <= 1"
             ^
     IndentationError: expected an indented block



In [11]: if len("this") < 4:
             print "what do you think len() does?"
         elif (3 + 4 - 6**2 < 0):
             print "7 - 36 < 0"

7 - 36 < 0
```

The `if` condition can be used independently of `elif` and `else`, much like in `C`. But if you use `elif`, you must use an `else`.

3.4.2 Loops

There are two main loop constructions in `Python`: `for` and `while`. The `while` loop operates much like its `C` counterpart does. It is given a boolean statement that, while true, executes the subsequent code.

```
In [12]: a = 0
         while a < 10:
             print a**2
             a += 1
```

```
0
1
4
9
16
25
36
49
64
81
```

The `for` loop, by contrast, operates differently than the C version. In C, there are three parts

```
for (loop initialization ; condition ; update)
{
    code;
}
```

In this sense, `for` is structurally identical to `while`; it just is set up differently. In `Python`, `for` takes some sort of *iterable* type and iterates through it:

```
In [13]: for name in ['Adam', 'Amy', 'Alex', 'Alfonzo', 'Adrianne', 'Abbie', 'Al']:
             print name, "is cool"

Adam is cool
Amy is cool
Alex is cool
Alfonzo is cool
Adrianne is cool
Abbie is cool
Al is cool
```

The variable `name` cycles through the list of names, printing it in each step of the loop. This is essentially the same as `Java`'s enhanced `for` loop, and you will learn to love it. If you don't, and you just want to have an index `i` to go from 0 to 9, you can use

```
In [14]: for i in range(10):
             print i

0
1
2
3
4
5
6
7
8
9
```

This achieves the same effect as the C `for` loop.

3.5 Functions

In `Python`, functions are easy to program. The general syntax is as follows:

```python
def foo(args):
    [do stuff]
    return bar # optional
```

Simply pass the arguments you want to include into the function, do whatever you want inside the function (don't forget the whitespace!), and `return` as you would in `C`.

3.5.1 Try it!

1. Write a function, `print_args`, that prints each argument passed to it in one line.
2. Write a function, `list_float_to_int`, that converts an integer list into a float list.
3. Write a function, `centered_average`, which returns the arithmetic average of a list, excluding the largest and the smallest elements.

4 Lab Activity: The Bisection Method

Those of you who have taken a few computer science courses may have learned about something called the "Binary Search", which allows you to quickly find an element of an unordered list. We will extend this idea to allow us to find roots of continuous functions over a closed interval.

The idea for this procedure comes from the Intermediate Value Theorem: if you have a continuous function $f(x)$ on an interval $[a, b]$ for which you know that $f(a) < 0$ but $f(b) > 0$, then there must be some intermediary point c satisfying $f(c) = 0$. The Bisection Method provides us with a procedure for finding such a c. Here's how it works:

1. Start with a function $f(x)$ on an interval $[a, b]$ that you know it is continuous over.
2. Evaluate $f(a)$ and $f(b)$. Figure out which direction is positive and which one is negative. (Let's assume $f(a) < 0$ and $f(b) > 0$ for this description).
3. Choose the point $c' = \dfrac{a+b}{2}$, and evaluate $f(c')$.
4. If $f(c') < 0$, then you know that the root of the function must be to the right of c', so set $a = c'$ and repeat this search.
5. If $f(c') > 0$, then you know that the root of the function must be to the left of c', so set $b = c'$ and repeat this search.
6. Continue until you reach an arbitrary precision of your choice (or terminating after a finite number of steps).

Your task is to implement this function in `Python`. Use this to come up with an approximation of π. *Bonus*: Store an array to keep track of all of your c' values.

4.0.2 Our solution

```python
from numpy import sin
def bisection_method(ufunc, a, b, N):
    err = 1e-20
    if ufunc(a) > 0.: # forces f(a) to be a lower bound
        temp = a
        a = b
        b = temp
    c = 0.5 * (a + b)
    fc = ufunc(c)
    cvals = [c]
    for i in range(N):
        if -err < fc and fc < err: # if fc is within an error tolerance of 0
            return c
        elif fc < 0.:
            a = c
        else:
            b = c
        c = 0.5 * (a + b)
        cvals.append(c)
        fc = ufunc(c)
    return c, cvals
```

Alternatively, you can implement this recursively.

```
def bisection_method_rec(ufunc, a, b, N):
    err = 1e-20
    if ufunc(a) > 0.:
        return bisection_method_rec(ufunc, b, a, N)
    c = 0.5 * (a + b)
    fc = ufunc(c)
    for i in range(N):
        if abs(fc) < err or N == 0:
            return c
        elif fc < 0.:
            return bisection_method_rec(ufunc, c, b, N-1)
        else:
            return bisection_method_rec(ufunc, a, c, N-1)
    return c
```

```
In [15]: def bisection_method_rec(ufunc, a, b, N):
             err = 1e-20
             if ufunc(a) > 0.:
                 return bisection_method_rec(ufunc, b, a, N)
             c = 0.5 * (a + b)
             fc = ufunc(c)
             for i in range(N):
                 if abs(fc) < err or N == 0:
                     return c
                 elif fc < 0.:
                     return bisection_method_rec(ufunc, c, b, N-1)
                 else:
                     return bisection_method_rec(ufunc, a, c, N-1)
             return c
```

```
In [16]: print bisection_method_rec(lambda x: x*x - 3., 0., 2., 100)
```

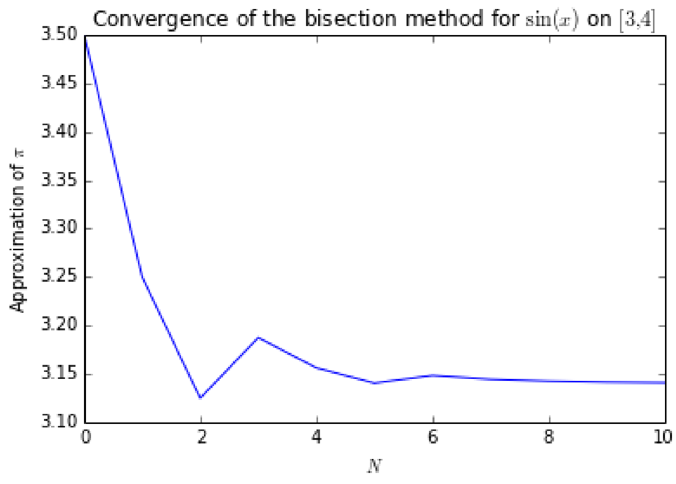
```
1.73205080757
```

```
In [17]: from IPython.display import Image

         Embed = Image("convergence.png")

         Embed
```

```
Out[17]:
```



5 Some advanced topics

Here's just a brief overview of the more advanced things you can do in `Python`. We won't use them in class, but it's good to know about them.

5.1 Classes

`Python` can be object-oriented like `Java`. You declare classes as follows:

```
class My_Class:
    self.field_one
    self.field_two

    def __init__(args): # Equivalent to constructors in Java
        # Initialization routine
        # Every class needs one

    def method_one(args):
        [do stuff]
```

Object-oriented programming favors modularity, bringing wonderful returns to scale. In `Python`, class fields and methods must always be specified with the `self` keyword, similar to `this` in `Java`. Unlike in `Java`, this is not optional in `Python`. Method overriding is also not supported in `Python`.

5.2 Lambda functions

`Python` can be functional like `Scheme`. A lambda function uses the following syntax:

```
lambda x: x**2
```

For example, instead of including a predefined function in `bisection_method`, you could use a lambda function. Lambdas and other household names for the functional programmer are built in, such as: * `map` * `filter` * `reduce`

If you like Scheme, you can make `Python` as functional as you want.

5.3 Comprehensions

In `Python` one important and useful tool is called list (or other Iterable) *comprehensions*. They allow you to construct lists from following rules, without the amount of code overhead that you might see in `C` or `Java`.

For example. Suppose you want to construct a list of all nonnegative even integers less than 20. The naive approach is to use some sort of loop:

```
evens = []
for i in range(20):
    if i % 2 == 0:
        evens.append(i)
```

Comprehensions allow you to take this routine and make it more concise. Instead, you can write

```
evens = [i for i in range(20) if i % 2 == 0]
```

For those of you who are comfortable with simple set theory, this is equivalent to saying

$$\text{Evens} = \{i \in \mathbb{N} : i < 20, i \equiv 0 \bmod 2\}$$

Comprehensions are a great way to make your code readable and concise. They cost a bit more (in terms of microseconds), but often the philosophy of `Python` is that readable code is preferable to slightly-faster ugly code.