SISMID Module 9
Lab 2: Introduction to Python

Thomas J. Hladish, Joel C. Miller and Lauren Ancel Meyers

July 2017

Why Python?

- it's relatively easy to learn and very easy to read
- it can be applied to a broad range of programming tasks
- Python is popular, so there is a lot of support online
- it's named after Monty Python—Really.

These qualities (less so the last one) make Python good for rapidly transferring ideas into working programs. When shouldn't you use Python? When it isn't fast enough: C/C++, Java, and Fortran are orders of magnitude faster, although they require much more development time and expertise.

Install Python

For this course, we will be using the Enthought Canopy Express, available for free here:

`https://store.enthought.com/downloads/`

I will be using Canopy v2.1.3 with Python 2.7. If you already program in Python and want to use Python 3.5, feel free, but know that you may have to do some code translation. Other versions of Canopy should also work with modest changes to the instructions herein. Your operating system (or platform) is probably 64-bit if you aren't sure.

If you are an academic user, you have the option to use the full version of Canopy for free by following the instructions at the bottom of that page. For this course, it does not matter which you use.

While all of the necessary software we will be using is free, open-source software, it can be tricky to get the libraries we need installed correctly on a class full of computers. Enthought Canopy solves this problem by providing a nice installer for a large number of major open source Python projects.

If you happen to be using Linux, you can use the linux installer for Enthought Canopy, or you can install everything you need from your distribution's repositories, which may be a cleaner solution. If you have any questions, please talk to us, and we can get you set up.

1 Using IDLE and IPython

Python is somewhat unusual among major programming languages in that it can be used interactively. This means that you can write and execute Python code line-by-line, somewhat like you use a calculator to interactively do math: rather than write all the expressions beforehand and then hit "RUN", you type them as you go. It is also possible to write an entire program in a text file and then run it, but we will start with an interactive environment.

The interactive exercises that follow were written using Python's Interactive DeveLopment Environment (IDLE) but can be done using either IDLE or IPython. The most obvious difference is that the IDLE prompt looks like "`>>>`" whereas the IPython prompt looks like "`In [1]:`". Navigate to your system's applications menu and click Enthought Canopy, then select "Editor" to begin. You can immediately start typing expressions at the prompt, or even writing a (small, simple) program. Type `2+5` and hit `<ENTER>`.

Try these, but feel free to also make up your own. Don't type "`>>>`"—that's just the IDLE shell prompt.

```
>>> 30/5
>>> 30/4
>>> 30.0/4
>>> 5*9
>>> 2**4
>>> 15-2**3
>>> (15-2)**3
>>> 2**10000
>>> 1.9**100
>>> x = 2; y = 3
>>> x**y
```

Notice four things:

- If you only use integers, Python will disregard the decimal portion throughout the calculation
- Python has a specific order of operations, which you can change using parentheses
- Python is comfortable with large numbers (very large for integers, up to about 10^{308} for floating-point numbers)
- After assigning values to variables, you can use them as you would numbers

IDLE will also let you use built-in functions, and even define your own. For now, let's look at a built-in function:

1.1 The `range(start, end, step)` function

You will find that you frequently need to loop through or store a sequence of numbers that are evenly spaced on the number line (such as all integers, all evens, all multiples of some arbitrary integer). `range()` solves exactly this problem. Some examples:

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(2,10)
[2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,-5, -1)
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
>>> range(3,16,3)
[3, 6, 9, 12, 15]
```

Try to come up with solutions to these:

- Generate the sequence of all odd numbers between 0 and 100 (e.g. 1, 3, 5, ... , 99)
- Generate the sequence of all even numbers from 100 to 0 (e.g. 100, 98, 96, ... , 0)

Functions in Python are always invoked using the function name, followed by parentheses. Depending on the function, you may need to specify function inputs, called *arguments*, inside the parentheses. The value you get back after the function executes is called the *return value*. In the case of `range()`, the return value is actually a list.

Just as you can store single values in a variable, you can store lists:

```
>>> x = range(2,7)
>>> x
[2, 3, 4, 5, 6]
>>> x[0]
2
```

In this case, `x` holds a list, and we can access individual elements of a list using the syntax `x[i]`, equivalent to $\mathbf{x}_i$ in standard mathematical notation. **Notice that list indexing starts with zero!** This is a very important point, and is the source of many programming bugs. Similarly, the index of the last element of a list is one less than the length of the list.

Note that `range()` can take, one, two or three arguments. If you leave off the *step* argument, `range()` assumes a default of `1`. **Also, `range(start, end)` uses the interval [start, end), so the end point is never included in the list.**

1.2 Browse command history with CONTROL-p, ALT-p, or UP/DOWN keys

If you get sick of retyping very similar commands, you can bring up recently typed commands using `CONTROL-p`, `ALT-p`, or the UP arrow key. Unfortunately, this is system-specific, so if one doesn't work, try another. `CONTROL-n`/`ALT-n`/DOWN will move forward in the command history.

1.3 Help!

You can also use the Python interpreter to find out how to use a certain function (although Google may be useful for finding examples—just serach for "python" and the function name). Try entering `help(range)`.

2 Example: Hello World

We can only get so far using an interactive environment. The Python shell is best suited for very small tasks that you just need to do once. Instead, programming is normally done by writing a text file, saving it, and then executing or running it.

Most programming tutorials start the same way, by creating a program that produces the text "Hello, World!". With Canopy running, go to File>New>Python file to start a new document. Type the following text in the new document window (without the line numbers!):

```
1 #!/usr/bin/python
2 print "Hello, World!"
```

Then save it as a file named `hello_world.py`. The exact name isn't important, only that it ends in ".py" and is somewhat descriptive. Also, it's a good practice to avoid spaces inside file names—use an underscore ("_") instead.

Now you can run your program by going to Run>Run File or pressing `CONTROL-r`. You should see the text `Hello, World!` appear in the terminal.

The second line is probably self-explanatory, but what's going on with the first line? In Unix-based systems (linux and OS X), Line 1 tells the operating system what to do with the stuff that follows—namely, use Python to interpret it. Windows, on the other hand, ignores Line 1, and instead recognizes files by extension, in this case `.py`. In order to make it possible to run your code on other systems, follow the conventions for both Windows and Unix. This is not necessary in every instance, but it's a good habit that may save you trouble later; it also visually distinguishes a program from an arbitrary chunk of code in a file.

3 Example: Averaging two numbers

Let's do something slightly more complicated. Many programs need to get some kind of input from a user; here we will ask the user to provide two numbers, which we will then average.

```
1 #!/usr/bin/python
2
3 # Normally we could do something like this:
4 num1, num2 = input("Enter two numbers separated by a comma: ")
5
6 average = (num1 + num2) / 2.0
7 print "The average is:", average
```

3.1 Variables and Assignment

There are a number of things going on in Line 4. First, a single equals sign (`=`) means *assignment*—in other words, evaluate the thing on the right-hand side (**RHS**), and store it in the variable(s) on the left-hand side (**LHS**). Programming deviates from math in that the RHS is always evaluated first—so it's perfectly legit to say

```
x = x + 1
```

assuming x is a number (the value of `x` increases by one).

In this case, the `input()` command is returning two values on the RHS, so the first one gets stored in the first variable on the LHS, and the second one in the second variable. Line 9 works similarly.

3.2 input() and raw_input()

Python makes getting user input extremely easy compared to most languages. As you can see, you type something in quotes that will be displayed to the user, and whatever the user enters is the return value of `input()`. Note that it's important for the user to behave as expected—if `input()` gets something other than two numbers separated by a comma, the program will fail.

`raw_input()` is similar to `input()`, except that Python does not try to process what the user has entered; instead, the input is stored as a single string.

4 Example: Reading in a stored network

I will provide you with an edgelist file (`simple_edges.txt`) that contains data that looks something like this:

```
0 15
0 5
0 14
0 7
1 9
1 2
1 10
[and so on]
```

Each line in this file represents an edge in a network with 20 nodes. The two integers on each line are the node numbers (unique identifiers); in other words, looking at the first four lines, we can see that Node 0 is connected to Nodes 15, 5, 14, and 7. To keep things simple, let's assume this is an undirected network, so saying there is an edge going from Node 0 to Node 15 is the same as saying there is an edge going from Node 15 to Node 0.

For now, let's just read in the file and write it out to the terminal. I've added some comments (bits of text that aren't executed) to this program to help explain how it works. Comments in Python are denoted using "#".

```
#!/usr/bin/python

file_in = open('simple_edges.txt') # let Python know you will want to access the file

for edge in file_in:        # loop through the file, storing each line in turn in 'edge'
    print edge              # print each line in turn, to make sure everything's working

file_in.close()             # Tell Python you no longer need access
```

This program looks very simple, but it presents some important concepts.

4.1 `for` loops

Computers excel at mind-numbingly repetitive tasks. By using loops, we can make the computer do very similar tasks an indefinite number of times with (presumably) different input each time. In this case, what's changing is the edge; later we'll replace the `print` statement with more interesting code. The general syntax of a `for` statement is

`for` *temp_variable* `in` *list_variable*:
block of code to execute each time

Since file variables are not technically lists, reading in a file in this way is something of a convenient idiom in Python.

4.2 Indentation

Indentation in Python is very important. In many languages, different blocks of code are set apart using special characters, often curly brackets. In Python, you are inside of a `for` loop as long as each line is indented the same amount. Just how much doesn't matter—it could be one space, it could be 18; what matters is that you are consistent within that block of code.

5 Example: Converting file types

Let's spend a bit more time looking at how to get data in and out of our programs, since this is such a fundamental part of doing research. It is possible to directly access and manage databases from within Python, but for now we will focus on the more ubiquitous task of working with files.

As demonstrated in Example 4, to work with a file within a Python program, we create a file variable using `open()`. The file variable type is much more abstract than the variable types up to now (it's not a number or text string or anything else we've looked at), but it's very easy to use.

5.1 Use `file.write(`*string*`)` to write to a file

Python is a great language for handling tasks like changing a file format. In this example, we will convert the original space-delimited edge file to a comma-delimited file called "simple_edges.csv" ("csv" stands for "comma separated values"). To create an output file, we still use the `open()` function, but this time we add a second argument: `"w"`. This time we'll name the file variable `file_out` to make it clear that it's a file variable opened for output.

```
1  #!/usr/bin/python
2
3  file_in = open('simple_edges.txt')         # Input file
4  file_out = open('simple_edges.csv', 'w')   # "w" is for write--be careful, since
5                                             # this can overwrite an existing file!
6
7  for edge in file_in:                  # Same as before
8      node1, node2 = edge.split(' ')    # Split the line on spaces
9      csv_edge = node1 + ',' + node2    # Concatenate the parts together, comma-separated
10     file_out.write(csv_edge)          # Write the line to file_out
11
12 file_in.close()       # Closing an input file isn't usually important
13 file_out.close()      # Closing output files is!  Otherwise you may lose data
```

On Line 8, I used the *string*`.split()` function. The syntax might seem odd at this point, since the variable to be split goes in front of the function name instead of inside of the parentheses, but otherwise it's as simple as it looks: if `line` consists of text separated by spaces, `split()` will break that string into "words" (numbers, in this case), discarding the space, and in this case the newline (the "return" character) as well.

On Line 9, we concatenate three strings together; we will look at this and similar operations in Section 7.1.

6 Example: Evens and Odds

In this example, we will look at a program that generates a list of integers, and then identifies whether they are even or odd.

```
1  #!/usr/bin/python
2
3  for num in range(10):        # loop through [0, 1, ..., 8, 9], storing each number in "num"
4      if num % 2 == 0:         # check to see if num modulo 2 (the remainder) is zero
5          print "Even:", num   # if so, label it as an even number
6      else:                    # in all other cases
7          print "Odd:", num    # label it as odd
```

Here again we see the `for` loop, this time looping through the elements of a list, rather than the lines of a file. Just as indentation denotes that you are inside a `for` loop, the same holds true for `if` statements. In this case, the `print` statement on Line 5 is nested inside the `if` block starting on Line 4, which is nested inside the `for` loop starting on Line 3.

Assignment and Equality

We've already seen how a single equals sign (`=`) means assignment; here we see that two equals signs (`==`) means an *equality test*, or "Are these two things equivalent?" The opposite of (`==`) is (`!=`), read "does not equal". Take a moment and explore this in IDLE. Try some of these:

```
>>> 5 == 6-1
>>> 5 == 6-2
>>> 5 != 6
>>> x=2; x == 2
>>> x == 3
>>> True == (x==2)
>>> False == (3==x)
```

Conditionals

The `if ... else` here is an example of a control structure that uses a *conditional*, in this case whether `num` is divisible by 2. `if ... else` constructions are what you need whenever you want to execute different code, depending on whether something is true (that's the *conditional* statement).

7 Example: Determining degree sequence and distribution

In the next lab, we will look at how to use NetworkX to do network-based calculations and simulations. Using the Python we have discussed so far, however, we can already determine the degree sequence, and therefore the degree distribution for a network.[1]

We can represent the degree sequence as a list; all we have to do is count the edges for each node. Since the node names in `simple_edges.txt` are consecutive integers from 0 to 19, we can use the node names as list indices. Each position in the list will then contain the number of edges connected to that node/index.

There is one caveat: even though *we* know that the node names are integers, Python assumes everything read in from a file is text—so we have to explicitly say that they are integers (see Line 10 below).

```
1  #!/usr/bin/python
2
3  file_in = open('simple_edges.txt')
4
5  net_size = 20                        # I know that the file will have 20 nodes
6  deg_seq  = [0] * net_size            # Create a list of 20 zeros using list multiplication
7
8  for edge in file_in:
9      node1, node2 = edge.strip().split(' ')    # Remove the newline, then split on spaces
10     node1, node2 = int(node1), int(node2)     # Convert the names to integers
11     deg_seq[node1] = deg_seq[node1] + 1       # Increment the edge count for node 1
12     deg_seq[node2] = deg_seq[node2] + 1       # and node 2
13
14 print deg_seq                         # Let's look at what we've got
15 file_in.close()
```

7.1 String and list addition and multiplication

On Line 6, we see an example of list multiplication. Addition, or more properly *concatenation* is also defined for lists. Since these operations work in the same way for strings, here are examples of that as well:

```
>>> 'abc' + 'de'          # string concatenation
'abcde'
>>> 'abc' * 3             # string multiplication
'abcabcabc'
>>> [0, 1] + [2, 3, 4]    # list addition
[0, 1, 2, 3, 4]
>>> [0, 1] * 4            # list multiplication
[0, 1, 0, 1, 0, 1, 0, 1]
```

7.2 Variable types and type conversions

So far, we've used three types of single-value variables: integers, floating-point numbers, and strings. Others do exist, but you will rarely need to use them. In the following example, we create one variable of each of these types.

[1] Remember that a node's *degree* is the number of edges connecting it to other nodes; a network's *degree distribution* is therefore the distribution of these values for all nodes in the network.

```
>>> a = 3
>>> b = 3.14159
>>> c = '3.14'
>>> a, b, c
(3, 3.14159, '3.14')
```

Python automatically recognizes that **a** is an int, **b** is a float, and **c** is a string. Note that integers are stored precisely, floats are stored approximately, and strings are quoted.

Sometimes it is necessary to convert from one type to another, for example when we have integers but we are doing normal division, or as in Section 7 where we needed strings to be treated as integers. As long as the input is sensible, you can convert from any type to any other type:

```
>>> float(a), float(c)
(3.0, 3.14)
>>> str(a), str(b)
('3', '3.14159')
>>> int(b), int(float(c))  # int(c) doesn't work, because 2 conversions are needed
(3, 3)
```

7.3 Visualizing the degree distribution

If your degree sequence program is correct, you should have gotten the following output in the terminal:

```
[4, 4, 3, 7, 2, 4, 4, 4, 3, 5, 3, 6, 8, 4, 5, 5, 3, 1, 4, 5]
```

In order to visualize that list as a histogram, we can use the `hist()` and `show()` functions in the `pylab` library. Add the following lines to the end of your degree sequence program from above:

```
# ... code from degree sequence program ...

from pylab import hist, show        # import these two functions from the pylab library
hist(deg_seq, bins=8, range=(1,9))  # hist(deg_seq) will work, but not look very good
show()                              # display the plot; necessary with some pylab versions
```

The network is small, of course, but you probably noticed that the distribution is roughly bell-shaped. This network was generated using the Erdős-Rényi algorithm, so the underlying distribution is poisson.

Additional exercises

1. Write a program to calculate the volume and surface area of a sphere from its radius. Ask the user for the radius using `input()`. Use the following formulas:
   $V = \frac{4}{3}\pi r^3$
   $A = 4\pi r^2$
   You can approximate π as 3.1415926, and in Python, 2^3 is written `2**3`.

2. Write a program that calculates the squares of the first n natural numbers (1, 2, 3, 4, ...), where n is provided by the user. Use the `input()` function for this.

3. Write a program that prompts the user for two numbers, and reports whether the second number is a factor of the first.

4. Write a program to sum a series of numbers entered by the user. The program should first prompt the user for how many numbers are to be summed. It should then input each of the numbers and print a total sum.

5. Write a program that will determine, for each integer between 1 and 100, whether that number is prime. Print the output for the user. Can you make it faster?

Other resources

You will find that Google is enormously helpful in learning Python. Try some of these resources I found:
Online:

- `http://wiki.python.org`
- `https://developers.google.com/edu/python/`
- `http://anh.cs.luc.edu/python/hands-on/handsonHtml/handson.html`

Books:
Python web resources are so good that you may not need a book. If you're interested, however, I highly recommend **Python Programming: An Introduction to Computer Science** by John M. Zelle, although it is weighted more heavily toward teaching computer science than a Python reference *per se*. Also consider **The Quick Python Book** by Naomi R. Ceder, **Python Essential Reference** by David M. Beazley, and **Python Scripting for Computational Science** by Hans Petter Langtangen.