BCH364C/394P Systems Biology/Bioinformatics Spring 2017 Marcotte
A Python programming primer

Python: named after *Monty Python's Flying Circus*
(designed to be fun to use)

Python documentation: http://www.python.org/doc/ & tips: http://www.tutorialspoint.com/python
Good introductory Python books:
Learning Python, Mark Lutz & David Ascher, O'Reilly Media
Bioinformatics Programming Using Python: Practical Programming for Biological Data, Mitchell L. Model, O'Reilly Media
There are some good introductory lectures on Python at the Kahn Academy:
https://www.khanacademy.org/science/computer-science
& Codeacademy: http://www.codecademy.com/tracks/python
A bit more advanced: *Programming Python*, 4th ed., Mark Lutz, O'Reilly Media

Although programming isn't required to do quite a bit of bioinformatics research, in the end you always want to do something that someone else hasn't anticipated. For this reason alone, if for no other, I'd recommend learning how to program in some computer language. For bioinformatics, many scientists choose to use a scripting language, such as Python, Perl, or Ruby. These languages are relatively easy to learn and write. They are not the fastest, nor the slowest, nor best, nor worst languages; but for various reasons, they are well-suited to bioinformatics. Other common languages in the field include R and perhaps C/C++ and Java.

If you think only about handling biological data, it tends to be on the extensive side. For example, the human genome is about 3×10^9 nucleotides long, so even at only 1 byte per nucleotide (i.e., letter), this runs to about 3 GB worth of data. In one of our various databases in the lab, we have about 1300 fully sequenced genomes, encoding about 4 million distinct genes. These are mostly bacterial genomes, which are smaller, so all of this takes up a bit under 10 GB worth of disk space. Other of our various collections of data occupy ~60 TB (terabytes) of disk space. Obviously, handling data in a convenient and fast manner is often a practical necessity. A typical bioinformatics group might store its data in a relational database (for example, using the MySQL database system, whose main attractions are that it is simple to use and completely free) and will do most analyses in Python, Perl, R, or even C++. We won't spend time talking about MySQL, C++, etc., but will spend the next few lectures giving an introduction to Python. This way, you get (1) at least a flavor for the language, and (2) we can introduce the basics of algorithms.

Starting with some example programs in Python:
Programs in Python are written with any text editor. If you really wanted to, you could program one in Notepad or Google Docs, save it as a text file, then run it on a computer that has the Python compiler. We're going to be using the Python Integrated Development Environment (IDE) that you installed in the first homework assignment on Rosalind. The filename (at least on Windows machines) is called "IDLE (Python GUI)". Open this, and let's start exploring.

A Python program has essentially no necessary components. So, a very simple program is:

```
print("Hello, future bioinformatician!")      # print out the greeting
```

That's it! Type this into your IDLE Python shell (a "shell" in computing is a user interface, often command line-based, i.e. you type in commands at a prompt).

The command you just typed in will be executed and the output in the IDE looks like this:

```
Hello, future bioinformatician!
```

Rather than just type in sequential commands, we can write an entire program and save it to be run later. In IDLE, you can do this by opening a new window (File -> New Window), then typing in your program into the new window and saving it (File -> Save As). Let's call it `hello.py` . (The names of Python programs traditionally end in `.py` .) You can then run the program (Run -> Run module), and in the main Python shell, you should again see the results of your program.

Notice the comment after a pound sign. Python ignores everything written after a pound sign, so this is how you can write notes to yourself about what's going on in the program. The only real command we've given instructs Python to write (`print()`) what you have placed between the quotes on the computer screen.

Let's try a slightly more sophisticated version:

```
name = raw_input("What is your name? ")  # asks a question and saves the answer
                                         # in the variable "name"
print("Hello, future bioinformatician " + name + "!") # print out the greeting
```

This is a bit more complex. Type this in & save it as `hello2.py`. When you run it this time, the output looks like:

```
What is your name?
```

If you type in your name, followed by the enter key, the program will print:

```
Hello, future bioinformatician Alice!
```

So, we've now seen one way to pass information to a Python program. Going through the program line by line shows:
Line 1: This is a specialized Python command called `raw_input`, which prints a line without a newline, and then saves what you type into a variable called `name`. Note that if you wanted it to print a newline, you could do `name = raw_input("What is your name?\n")` . The `\n` indicates a new line.
Line 2: Another note to ourselves
Line 3: Lastly, we print out the message, but this time with your name included. Any variable can be printed in this fashion, by simply including it in a print statement.

Okay, so now we've seen two very simple Python scripts. Quite a few programs can be written that simply read in and print out information. Although we read in information from the keyboard (e.g. when you typed your name in), it's not much harder in Python to read it in from a file, so you can go a long ways with this general level of programming. However, we'd like to get to the point where we can do some calculations as well, so let's look at the main elements of programs, so that we can eventually write a program that actually does something more interesting than just printing or reading a message.

A note about versions:
Most bioinformaticians use Python 2.7. There are some subtle but important differences between Python 3+ and Python 2.7 which mostly won't matter to you. But if you have problems running the

scripts, you should make sure you're using 2.7. In the version I'm using to make this handout (version 2.7.3), you can verify this in IDLE (Help -> About IDLE).

Some general concepts:
Names, numbers, words, etc. are stored as *variables*. Variables in Python can be named essentially anything, as long as you don't pick a word that Python is already using (e.g., `print`). A variable can be assigned essentially any value, within the constraints of the computer, (e.g., `BobsSocialSecurityNumber = 456249685` or `mole = 6.022e-23` or `password = "7 infinite fields of blue"`).

Groups of variables can be stored as *lists*. A list is a numbered series of values, like a vector, an array, or a matrix. Lists are variables, so you can name them just as you would name any other variable. The individual elements of the list can be referred to using `[]` notation. So, for example, the list `nucleotides` might contain the elements `nucleotides[0] = "A"`, `nucleotides[1] = "C"`, `nucleotides[2] = "G"`, and `nucleotides[3] = "T"`. By convention, lists in most scripting languages start from zero, so our four-element array `nucleotides` has elements numbered 0 to 3.

Lastly, there's a VERY useful variation on lists called a *dictionary* or *dict* (sometimes also called a *hash*). Dictionaries in Python are groups of values indexed not with numbers (although they could be) but with other values. Individual dictionary elements are accessed like array elements. For example, we could store the genetic code in a dictionary named `codons`, which might include 64 entries, one for each codon, such as `codons["ATG"] = "Methionine"` and `codons["TAG"] = "Stop codon"`.

These are the three most frequently-used types. Now, for some control over what happens in programs. There are two very important ways to control programs: `if` statements and `for` loops.

`if` *statements*
Very often, one wishes to perform a logical test. This is usually done with an `if` statement or command. An `if` command applies Boolean logic to a statement, then performs a different task depending on the logical result. So, we might be looking at a DNA sequence and then want to ask:

If this codon is a start codon, start translating a protein here.
Otherwise, read another codon.

The Python equivalent of this might be:

```
if dnaTriplet == "ATG":
          # Start translating here.  We're not going to write this part since we're
    # really just learning about IF statements
else:
    # Read another codon
```

The logical test follows the `if` statement. If true, Python executes the commands on the lines following the `if … :`. If false, the commands following the `else:` (which is optional) are run. **Note that Python cares about white space (tabs & spaces) you use**; this is how it knows where the conditional actions that follow begin and end. **These conditional steps must *always* be indented by the same number of spaces (e.g., 4).** I recommend just using a tab (rather than spaces) so that you're always consistent. A nice consequence of this rule is that your code (and that from other people) will be highly readable.

The following logical arguments can be used:

`==` equals
`!=` is not equal to
`<` is less than
`>` is greater than
`<=` is less than or equal to
`>=` is greater than or equal to

and several others. Multiple tests can be combined by putting them in parenthesis and using Boolean operations, such as `and`, `not`, or `or`, e.g.:

```
if dnaTriplet == "TAA" or dnaTriplet == "TAG" or dnaTriplet == "TGA":
      print("Reached stop codon")
```

`for` *loops*

Often, we'd like to perform the same command repeatedly or with slight variations. For example, to calculate the mean value of the number in an array, we might try:

Take each value in the array in turn.
Add each value to a running sum.
When you've added them all, divide the total by the number of array elements.

Python provides an easy way to perform such tasks by using a `for` loop.

```
grades = [93, 95, 87, 63, 75]   # create a list of grades
sum = 0.0                       # variable to store the sum

for grade in grades:            # iterate over the list called grades
    sum = sum + grade           # indented commands are executed on
                                # each cycle of the loop.

mean = sum / 5                  # now calculate the average grade
print ("The average grade is "),mean   # print the results
```

`for grade in grades` tells Python that it should visit each element in the `grades` list, and should call that element `grade` for the current loop cycle. Lines which are indented following the `for` line are repeated during each cycle. This means that Python will perform the sum operation for each of the elements of `grades`. Note that Python will perform most mathematical operations, such as multiplication `(A * B)`, division `(A / B)`, exponentiation `(A ** B)`, etc. (Don't worry about it for now, but down the line, one of the really nice features of Python that you will be able to take advantage of is that it has very extensive mathematical capabilities built into it.)

One important thing to realize is that Python keeps track of whether numbers are to be considered integers or floating point (and for that matter, long integers and complex numbers as well); you can indicate to Python that you want floating point by first defining your variables in that manner (e.g., `X = 1.0` versus `X = 1`). In the program above, our first use of the variable sum set it to be a floating point number. Just for fun, type in the program and run it, then change the definition of sum to read

```
sum = 0
```

Notice the difference?

These are the two most important ways of controlling a Python program. Another very useful control mechanism is the *function*, but you'll have to learn these on your own!

Reading and writing files

The last, but perhaps the most critical, element of Python programming to introduce is that of how to read information in from files and how to write it back out again. This is really where scripting languages shine compared to other programming languages. The fact that bioinformatics programmers spend a great deal of their time reading and writing files probably explains the preference for languages like Python.

In general, files are read line by line using a `for` loop.

```
count = 0                               # Declare a variable to count lines
file = open("mygenomefile", "r")        # Open a file for reading (r)
for raw_line in file:                   # Loop through each line in the file
    line = raw_line.rstrip("\r\n")      # Remove newline
    words = line.split(" ")             # split the line into a list of words
    # Print the appropriate word:
    print "The first word of line {0} of the file is {1}".format(count, words[0])
    count += 1                          # shorthand for count = count + 1

file.close()                            # Last, close the file.
print "Read in {0} lines\n".format(count)
```

We start by declaring a variable named `count`, which we'll use to count the number of lines in the file. The `open` statement then signals Python to find the file named "`mygenomefile`" for reading (indicated by `"r"`) and assign it to the variable `file`. The `for` loop then kicks in. Each line of the file is read in, one at a time, and temporarily assigned to the variable `raw_line`.

In text files created in Mac and Unix/Linux computers, each line ends with a newline character (`\n`). In Windows, each line ends with a carriage return as well as a newline (`\r\n`). We often need to strip those characters off, which we can do using `rstrip("\r\n")` (which works whether or not there is an `\r`). We assign the result to our own variable, `line`. Next we use a fun method called `split` to divide up the line wherever there is a space (or whatever we put between the quotes in the split command) and assign each piece to sequential elements of a list named `words`. We then print the first word on each line by writing out the corresponding list element (remember, lists are always indexed starting at zero, so element number 0 is the first element). Note that variables can be indicated in the print statement by placeholders (e.g., {0}), and then called out specifically at the end of the line after the format command. Lastly, we increment our line counter `count` by one. The cycle continues until each line has been read, and after finishing, we close the file. Had we wanted to, we could have remembered every line, or perhaps only a few words from each line, by feeding in the lines (or words) into array elements as we read the lines in.

Not too hard! On to writing files. Actually, it's about the same as for reading files, but with a few small changes. In the `file` command, instead of using `"r"` for 'read,' use `"w"` for 'write.'
Then, every time you wish to write something to the output file, use the command file.write(some_variable) instead of print:

```
file = open("test_file", "w")
file.write("Hello!\n")
file.write("Goodbye!\n")
file.close()      # close the file as you did before
```

Note the presence of the newline character (`\n`) in the second and third lines. Without these, your output will look like `Hello!Goodbye!` Unless you specify otherwise, you can find the new text file you created (`test_file`) in the default Python directory.

So, our earlier file that prints the genetic code might be modified to write the code to a file:

```
file = open("myoutputfile", "w")        # open myoutputfile for writing
for triplet in codons.keys():           # loop through codons dictionary keys
    file.write("The amino acid encoded by {0} is {1}\n".format(triplet, codons[triplet]))
file.close()
```

You can actually open and write to multiple files at once. Just assign them to variables other than `file`.

Putting it all together

These are the most important basic ideas for writing functional programs in Python. If you can master these concepts, the rest will come with practice, necessity, and exposure to more advanced programs. So, to close, let's write a complete program to read in some real data, perform a calculation, and print the results. This will be a simple program to calculate the nucleotide frequencies in a DNA sequence.

```
seq_filename = "Ecoli_genome.txt"
total_length = 0
nucleotide = {}            # create an empty dictionary

seq_file = open(seq_filename, "r")
for raw_line in seq_file:
    line = raw_line.rstrip("\r\n")
    length = len(line)     # Python function to calculate the length of a string
    for nuc in line:
        if nucleotide.has_key(nuc):
            nucleotide[nuc] += 1
        else:
            nucleotide[nuc] = 1
    total_length += length

seq_file.close()

for n in nucleotide.keys():
    fraction = 100.0 * nucleotide[n] / total_length
    print "The nucleotide {0} occurs {1} times, or {2} %".format(n, nucleotide[n], fraction)
```

Let's choose the input DNA sequence in the file conveniently named `Ecoli_genome.txt` to be the genome of *E. coli*, which we can download from the **Entrez genomes** web site
`http://www.ncbi.nlm.nih.gov/nuccore/49175990?report=fasta` (also on our class web site)
The format of the file will be line after line of A's, C's, G's and T's, such as:
agcttttcattctgactgcaacgggcaatatgtctctgtgtggattaaaaaaagagtgtc... and so on.

Running the program produces the output:

```
The nucleotide A occurs 1142136 times, or 24.6191332553 %
The nucleotide C occurs 1179433 times, or 25.423082884 %
The nucleotide T occurs 1140877 times, or 24.5919950785 %
The nucleotide G occurs 1176775 times, or 25.3657887822 %
```

So, now we know that the four nucleotides are present in roughly equal numbers in the *E. coli* genome.