-- Introduction

Programming is a form of expression in which you describe structure and operations on that structure. Practically, and within the context of this workshop, much of programming in genomics surrounds ways to organize sequences such that you can effectively ask questions, and interrogate the data.

Programming is an incredibly powerful tool for data analysis as often, within a research project, questions arise that cannot be answered with the present software available. Having a cursory handle on how to use at least one language expands the utility of your computer infinitely, and in turn, widens the types of questions that, as a researcher, you can ask of your data. Also, programming is FUN!!!

Beyond this very light tutorial, Zed Shaw's Learn Python the Hard way (http://learnpythonthehardway.org/) is excellent and free. LPTH will go into much more depth than we can do within this tutorial. When working through this tutorial, or any tutorial on programming, it is essential to work through the exercises. Like a spoken language, you won't learn without using.

-- IPython

IPython is an extension to Python that provides a bit of a richer interactive environment. Recently, the IPython project pushed out their Notebook interface with the intent of facilitating reproducibility (e.g. http://nbviewer.ipython.org/) with an emphasis on the sciences. These notebooks can be saved and distributed later very easily, and essentially any content can be embedded. Further details and tutorials about the Notebook itself can be found here: (http://ipython.org/ipython-doc/dev/interactive/htmlnotebook.html).

Lets start the Notebook. From the command line in your Virtual Machine, type the following:

ipython notebook --pylab

This will produce a bunch of text that you can ignore, and additionally pop up a Firefox web browser. You're now in the IPython notebook.

The Notebook interface is intended to feel a bit like Google Docs, but for programming. First, create a new Notebook from the Dashboard by clicking "New Notebook." A new tab will open. Next, click the "Untitled0" text and give your Notebook a name.

The notebook is based on cells. Each cell can be executed in any order. Executing a cell makes the code and variables run within the cell available to the entire notebook. The cells can be executed by clicking either the play button or by pressing "shift + enter". For instance, type the following into a cell, and execute it:

```
print "Hello you!"
```

You're now a programmer. Time for champagne (http://xkcd.com/323/).

-- IPython again

A little bit more on IPython, the tab-key is yet again your best friend! We'll play with this in a second.

-- How to get help

Google is your number one ally. Often, you can search for the types of errors you encounter and get a direction to follow to resolve your problem. If that fails, www.stackoverflow.com is a free user-driven programming community. It is very easy to use and the responses are great. Additionally, the search interface actually works and you may be able to mine for your problem there. Last, www.seqanswers.com may be able to assist on problems that may be more bioinformatic in nature.

Python itself contains a heavy amount of documentation. This stems from the easy mechanisms to document code within the language. You can call help() on just about everything and generally get back useful information. Try the following:

```
help(1)
```

Who knew the number 1 was so detailed...

IPython Notebook extends the help() with tool tips. Type the following:

```
plot(
```

Don't put the closing parenthesis and wait a few seconds. A help box will appear describing the function and what to pass. This is a good place to start, at least for how to use a particular function.

-- Variables and Types

Variables hold information, like a string that represents DNA, or a floating point number representing G+C content, or maybe an integer that describes the number of sequences that came screaming off your friends HiSeq2000. Variables have a type that is associated with them. This is incredibly important within programming: the type is how the machine interprets the 1's and 0's that reside in the systems memory, and different types have different properties and constraints. Types are important for optimization as the language can make more efficient use of memory and low level processes if the types used match your problem well. Lets explore a few basic types that are available within Python (and many programming languages, for that matter).

The integer is the first one. This type represents whole numbers.

```
my_int = 5
type(my_int)
```

So what though? Since an integer can only be a whole number, what happens if you divide by 2?

```
print my_int / 2
```

That's a little annoying but important to realize this can lead to "fun" bugs in your program. If you instead divide by a floating point number, or one that has a decimal, Python will perform what is known as a type promotion where the number that is not a float will become one.

So what are these floats? Lets create one

```
my_float = 2.123
type(my_float)
```

How are floating point values represented in 1's and 0's? In short, a portion of the 1's and 0's of memory that holds the number describes the mantissa, and a portion describes the exponent. While this works very well, it is possible that floating point numerical errors can happen if you perform operations on very large numbers and very small numbers -- logarithms are your friend. As a consequence of how floating point numbers are represented, there is a limit to the resolution of the numbers. The smallest difference between two floating point numbers that can be represented on a computer is known as machine

epsilon. To see it, type:

```
from sys import float_info
print float_info.epsilon
```

Last, lets create a string. Strings are one way that a piece of DNA can be represented on a computer:

```
my_str = "aattggcc"
type(my_str)
```

Strings have some additional operators associated to assist pulling out substrings. Try the following:

```
print my_str[0]
print my_str[1]
print my_str[2]
print my_str[1:5]
```

Do the results make sense? Now try over reaching, or stepping out of bounds:

```
print my_str[10000]
```

The output you're looking at is known as a traceback. This helps the programmer debug by identifying the offending line of code. In a complex script, the traceback may contain many lines indicating the path of execution and the location of the failing piece of code. They always end with the specific exception that was thrown. Try Googling the exception (IndexError: string index out of range) to learn a little more.

For a an exercise, using a single print statement, print three lines such that the first line contains a string, the second line contains a float and the third contains an integer. (hint: a newline is \n)

Before moving, explore the string with dir() to see what methods are part of the string. Try to figure out what a few of the methods do, and see if you can use them. (hint: some_var.some_method())

-- Constructs

In some senses, programming can be conceptualized with a flow chart (http://xkcd.com/518/). Right now, we can add in lines into a single cell in

the notebook and execute them, however, the execution will only be linear. Do this, then this, then this. Pretty boring. What we really want is to be able to, say, walk over a piece of DNA (i.e. looping) or perhaps execute different code based on the presence of a particular k-word (i.e. if-statements). More generally, these constructs are forms of flow control. Lets first experiment with an if-statement. The idea is simple: if this happens, do this, otherwise do that.

```
if 5 > 0:
    print "that's a refreshing"
else:
    print "this language is broken"
```

The if-statement operates on boolean logic. If the conditional (the 5 > 0 part) evaluates as True, the block of code underneath the conditional will be executed. Try just typing 5 > 0:

```
print 5 > 0
```

Variables can be used as well. For instance:

```
length_of_dna = len("aattggcc")
if length_of_dna > 5:
    print "our DNA is over 5 nucleotides long"
else:
    print "our DNA is really short!!!"
```

In the preceding example, we used a Python built in function called len(). Before moving on, learn a little more about it and experiment with it. Does len() work on integers? How can we use a forloop if we have multiple conditions? (hint: elif)

We can now perform conditional operations. That's super cool, right? Think of all the things you could conditionally do now, like, drink a beer if the right one is available. But, what if there are a lot of different types of beer? And we need to evaluate all of them? For this, we need a way to loop. Try the following:

```
my_dna = 'aattggcc'
for nucleotide in my_dna:
    print nucleotide
```

We've now walked over our DNA. We could interpret the code above as: for each nucleotide in my_dna, store that character in the new variable nucleotide, and print the character. But what about the beer you ask???

```
for beer in ['lager','pale ale','pilsner','stout','porter']:
    if beer == 'pilsner':
        print "when in cesky, go pilsner or go home"
```

We've introduced a few new things here. First, what's this funky thing in the square brackets? It's known as a list, and we'll dive into lists in the next section (but please experiment!). Second, we introduced a new type of evaluation, the ==. A single equals sign is used to denote assignment, like when we stored the integer in my_int. Two equals signs is used to test whether two things are equal to each other. For instance:

```
print 5 == 5
print 5 == 6
```

A few other important boolean evaluations are: <, >, !=, >= and <=. What do they do?

There's one more important loop known as a while-loop. The structure is: while <conditional>. See if you can recreate either of the forloops above using a while loop instead of the forloop. (hint: replace <conditional> with some type of logical test)

Before moving to the next section, there are a few more interesting and very useful operators (beyond +, -, *, and /). Play with the following and figure out what they do: +=, -=, and %.

An important note that can cause confusion. Python cares about whitespace. Indentation tells Python that the scope has changed and defines a new block of code.

-- Complex types

We've played with the basics, now lets dive deeper. One of the absolutely fantastic things about Python is that it offers a few incredibly useful datatypes for use within the language. Sure, we could always implement complex data types (and it is done quite often in practice), but there are a few data types that are used so frequently that the designers of Python opted to just include them (and uber optimize them).

The first type is a list. A list is an ordered list of items and can be created by using the square brackets. For instance:

```
my_list = [10,20,'attggcc']
```

If you've programmed before, you may be shocked to see that the list can contain multiple different types. In this case, we have two integers and a string. Many languages do not allow this as type ambiguity is computationally expensive, however it allows for ease of programming (although there are strong arguments against this). Try indexing into the list and pull out the first item. You can also test whether an item is present in a list using the keyword "in". For instance:

```
print 10 in my_list
print 100 in my_list
```

However, when performing lookups on lists, Python has walk through the list and evaluate each item until it finds the item you're looking for (or doesn't find the item). This can be expensive if the list is large or if the lookup is performed repeatedly. A more formal way to describe this is that the lookup time for a list is linear to the size of the list. The fancy notation used here is called Big-O, and a list look up is O(N), where N is the size of the list.

The second type is called a set. Sets act like sets from math. They are unordered data structures that allow instantaneous lookup (very useful for searching); effectively O(1).

```
my_set = set([1,2,3,4,4,4,4,5,'aattggcc','ccggttaa'])
print my_set
```

Now try to pull out some items... but that doesn't make sense, does it? One common use for sets is to help with lookup. Try the following:

```
if 1 in my_set:
    print "1 is in my set"
if 10 not in my_set:
    print "10 is not in my set"
```

If you call dir() on your set, you'll see a lot of other functionality like intersections and unions.

Next is the Python dictionary. Dictionaries allow you to associate a key with a value, and allow instantaneous lookup like sets. Dictionaries are created using curly braces. For example:

```
my_dict = {'seq1':'aattggcc', 'seq2':'ttggttgg','seq3':'tgtgccc'}
```

The basic form is {key:value, ...}. You can add more items to the dictionary easily using square braces:

```
my_dict['another sequence'] = 'tatatat'
```

You can pull sequences out in a similar fashion. Experiment!

These types support whats known as iteration, which is a fancy way to say that Python knows how to walk over them with a forloop. Let's say we wanted to look at all of the keys we have in our dictionary:

```
for key in my_dict:
    print "The key is:", key, "And the value is:", my_dict[key]
```

Imagine the possibilities... for example, you could load a lot of sequences into a dictionary and pull out arbitrary ones to process easily. Or, perhaps you could store loci annotation information such that you could get back the start and end positions for proteins by querying this data structure.

One last type worth mentioning is the array offered by the package numpy. Let's add it to our environment and create an array. These objects are useful for numerical work, and we'll show some examples here:

```
from numpy import array
a = array([[1,2,3],[4,5,6]])
print a
print a + 10
print a.sum()
```

Stepping back slightly, it is important to understand that the choice of algorithms and data structures can have large impacts on your code. Say we wanted to determine if two words are anagrams of each other. Two words are anagrams if the letters in one word can be rearranged into the other word. "quite" and "quiet" are anagrams. Within the context of sequence data, perhaps we want to know if two sequences have the same composition. We could naively check if the words are anagrams by shuffling the letters and checking for

equality:

```
from random import shuffle
word1 = list("quiet")
word2 = list("quite")
while word1 != word2:
    shuffle(word1)
    print word1
```

Stepping through this, we're adding some additional functionality to our environment. "random" is a Python module that contains a lot of useful functionality. You can explore what's in there by importing it ("import numpy") and calling "dir(numpy)"" or, see if you can Google it :). Back to the code, we're casting each string into a list in order to be able to shuffle them. A string is immutable (Google!), and in short, we cannot modified the contents of a string. However, a list is mutable, and thus can be modified. The while loop will loop until the words are identical, therefore are anagrams. If they are not equal, word1 is randomly shuffled. What are some problems with this approach? Can you think of a better way to solve this problem?

-- functions

Abstraction is central computers. For instance, web browser abstracts out the complex interactions with the internet protocols when you type in a web address. Behind the scenes, the web browser may have some function called get_url() that accepts a web address, but as a user, you don't need to care about those details. Functions offer a means to abstract out blocks of code. In addition, they make it easier to verify that the small components of your program are correct and adds verbosity. For instance, say you wanted to compute GC content on a set of sequences:

```
my_dict = {'seq1':'aattggcc', 'seq2':'ttggttgg','seq3':'tgtgccc'}
for key in my_dict:
    gc = 0.0
    for nt in my_dict[key]:
        if nt == 'c' or nt == 'g':
            gc += 1
    print gc / len(my_dict[key])
```

But that's hard to read and interpret easily. What if instead we chopped out the code the computes GC into a function, and gave it a pretty name:

```
def compute_gc(sequence):
    gc = 0.0
    for nt in sequence:
        if nt == 'c' or nt == 'g':
            gc += 1
    return gc / len(my_dict[key])
```

We have added in a few new terms here, "def" and "return". "def" tells Python that we want to create a function. And "return" allows us to have the function give a result back. We can now rewrite our forloop from above in a much more elegant form:

```
my_dict = {'seq1':'aattggcc', 'seq2':'ttggttgg','seq3':'tgtgccc'}
for key in my_dict:
    print compute_gc(my_dict[key])
```

It is much easier to read the code and see what's going on now, right? Here's a challenge: write a function that takes as input a string and a number and returns the set of all unique k-mers (where k is the number passed in).

-- File I/O

So you've got your sequences in a file and you want to play with them. How? Writing robust parsers is outside of the context of this tutorial, and in the case of Python they aren't necessary (Google for PyCogent). However, basic file input/output is pertinent. Lets first open a file and write some data to it.

```
my_file = open('a_test_file.txt','w')
```

Note the 'w'. That specifies the mode to open the file with. Be careful though, the 'w' is destructive! Any file open with 'w' will destroy its contents. How would you open a file to append data to it?

Lets put some sequences in our file:

```
my_file.write('aattggcc\n')
my_file.write('aattc\n')
my_file.write('aattttaggcc\n')
```

The write method is naive, and you have to structure the output exactly as you want. We added the newline character so that there is a single sequence per line. Now, lets read the sequences back in:

```
ro_file = open('a_test_file.txt','r')
```

There are a few methods to read the file. What are they? How would you read each each individual line? You'll notice that if you read the file once, you can't read it again. Once you've read the file, the offset, or position in the file that you're at, is the end of the file. You can reset the position by either reopening the file, or calling:

```
ro_file.seek(0)
```

We use 0 here because we want to go to the 0th byte in the file. See if you can read the sequences into a set.

-- matplotlib

You can make pretty pictures in Python too!! Try the following:

```
plot([1,2,3],[4,5,6])
scatter([1,2,3],[2,4,6])
```

Google "matplotlib example". It is a remarkably powerful visualization library.

-- Additional exercises

* Determine sequence similarity based on shared k-word composition. unweighted? weighted?
* Stitch together reads, or, assemble into contigs
* Histograms of k-word frequency