A set of notes by Charlie **Spring 2017**

Notes on 'Python for Genomic Data Science'

Third Module of the Coursera Genomic Data Science Specialization from John Hopkins University

No prior knowledge expected here. Steps to programming:

1. Identify the required inputs.
2. Make an overall design for the program, including the inputs list.
3. What will the output be? a file? a printout? a figure?
4. Refine the specification by adding more detail, then write the code.

Another way to organize your thoughts is to write 'pseudocode'. One of the most productive environments for genomic data-science is **Python** - easy to learn and extremely powerful - a simple but effective approach to object orientated programming. It's 'interpreted' – this means that you dont have to compile it: the computer will instantly interpret one line of code at a time, if required. For compiled programs, you have to run it through a compiler to create a binary object which gets run: Python converts to binary 'on the fly'. Python is extremely 'interactive', portable (cross-platform), and extensible ('wrapper' functions for other languages') with a large range of inbuilt functions and is very scale-able (small programs to large functions). Python can be used as a simple calculator. For example: `5+5`. ** can be used for powers e.g. `10**2=100`. The order of precedence of operators is the same as in standard maths (i.e. multiplication takes precedence over addition). Numbers can be different types: i.e. `type(5)` is an int, but `type(3.5)` is a float. Complex numbers: `type(3+2j)`. However, DNA and are really just strings of letters. e.g. `atg` or `this is a codon, isnt it`. You can also use a backslash as an escape character i.e. `''This is a string isnt it''`. Strings can span multiple lines with tripple quotes. The newline character is `\n`. Other escape characters: tab: `\t`, backslash:`\\`, double quote: `\"`. Here are the basic string operators:

- `+` : concatenate strings.
- `*` : copy string (replicate).
- `in` : membership.
- `not in` : non-membership.

Without variables, you cannot do anything in Python - theyre storage conatiners for numbers, strings , etc. The equals sign assigns a value to a variable. for example: `codon= 'atg'` and `dnaseq='gtcgcctraaccgtatatat'`. The name associated with a value is called a 'variable' because its value can change. In general, give your variables a meaningful name, rather than 'a' or 'b' which have no meaning. Note: case sensitive. The underscore character helps add readability of variable names. Variable names cannot use 'illegal' characters (e.g. pound sign) or begin with numbers. One important thing that we do is search for substrings in a bigger string. *This is especially important in genomics*. We can index strings using things like `[x]` and slice them with an operation like `[x:y]`. Note: we begin indexing things with 0. You can also use negative numbers while indexing to start from backwards. When we give two numbers in a slice, e.g. `[0:3]` will pick out a range of the first three characters of our string. If we leave off the second number, we index to the end of the whole string. `len` tells us the length of a string and `type` tells us the type of a string. We can find help about any built in function using `help()` which tells you everything that function () can do. Object orientated programming tells you about the objects, rather than the actions. The `string.count(str)` function tells us how many times `str` features in `string`. We can also case variables using `string.upper()` and `string.lower()`. The `string.find(str)` tells us what position `str` is in within `string` (but only the first occurance). We can reverse find using `string.rfind(str)` which will look from the end of the string. We can get booleans (True/False) with `string.isupper()` and `string.islower()`. We can also use other functions, such as `string.replace.()`

For our first example, we want to compute the GC content of a DNA sequence: the percentage of CGs in a sequence which is all Cs,Gs,As and Ts. Use the `string.count()` method and `str.len()`. To execute all

commands at once: put them all into a .py file, and execute the file (might need `#/usr/bin/python` ?). Make the file executable with `chmod a+x`. An important principle of all programming is to 'comment' your programs: what was it that you had in mind when you wrote the code? Everything inbetween tripple quotes is ignored by Python, as is everything after a hashtag at the end of the line.

We can capture raw input from the user using the `'input'` function, e.g: `dna=input('enter a dna sequence')`, where whatever the user types is then captured to the variable `dna`. This will always return a string (although you can convert it later, of course). Some string conversion functions:

- `int(x,[base])`: convert to an integer
- `float(x)`: convert to a floating-point
- `complex(real,[,imaginary])`: convert to a floating-point.
- `str(x)`: converts to a string.
- `char(x)`: converts to a character.

Don't forget! You can use formatting command when using enhanced `print.()` commands. For example: `print('The DNA sequences GC content is %5.3f %%' % x)`. There are a range of different formatting commands (e.g. % d, % 3d, % o, %e etc). One type of structure is a `list`: an ordered set of values: `geneexpression=['gene','5.16','0.0001512']`. We can alter values of an element of a list, print them out, slice them, etc. There are a whole host of operations on lists, such as concatenation and list. The `del` operation can be used destructively: e.g. `del list[1]` will delete the second element of the list and we can also count the number of times an element appears in a list and reverse it with list.reverse(). We can also `append` and `pop`, which allow us to treat list as 'stacks'. `append()` adds an element to the end of the list. The difference between append and extend is that append takes one element, but extend adds a second list to the end. The `list.pop()` function removes the last element added to the list. There are two ways to sort a list: `sorted()` function and another method is `mylist.sort()`. However, note that they do slightly different things. **Don't forget! help(list) and help(string) for all appropriate functions!** A `tuple` is like a list, but it is *immutable*. For example: `t=1,2,3`. We can surround them with or without parentheses. Another data structure is a `set`: an unordered collection with no duplicate elements (because no order, they have no index). Sets support mathematical operations like unions (|), intersections (&) and differences (-). Another important data type is the `dictionary`: which stores a *pair*: a key and value pair. This is especially useful for genomic data science: such as keys for sequences. We create a dictionary like this, wrapping the dictionary in {:

exampledict={'key1':'value1','key2':'value2','key3':'value3'}

Then, to gets its value out, index on the key, i.e. `exampledict[key1]`. You can do boolean tests of whether specific things are in your dictionary. We can also delete a key from the dictionary using `del exampledict[`key1]`. len(dict)` tells us the length of our dictionary (the number of key-value pairs), and `list` returns all of the keys in the dictionary, and `list.values()` gives you all values (as opposed to keys). We can also sort on keys and values. Here is a comparison summary of the sequence data types:

Control statements allow non-sequential execution of code. `if` and `else` are extremely common examples which occur in the majority of programming languages. The keyword `if` is followed by a condition. The condition is a *boolean*: a true or a false. If `True`, the code is executed. Boolean are formed with the help of *comparison* (e.g. ==), *identity* (e.g. `is` and `is not`) and *membership* (e.g. `in` and `not in`) operations. If the results of these statements are `False`, then we need an `else:` statement to determine what to do next. The `else` statement must be at the same level of indentation. We can have multiple sequential conditions with `elif:`. We can nest multiple conditions into one boolean with 'logical operations': 'and', 'or', and 'not'. For example: `if (X and Y) and not Z:`.

Loops are one of the most fundamental tools in any programming language. Two types: `while` (do something while a condition is true) and `for` (do something iteratively across a list or over a range). Again, note, that you have to be careful with *identation*! Examples: `for s in list:` or `for i in range(4):`. A genomic example: `for each character in proteinseq:`, `if protein[i] not in protein`, do something. We can break out of a loop prematurely using the `break` command: this leaves our loop entirely. The `continue` statement jumps out of the loop and doesnt finish iterating. The `pass` statement is a *placeholder* - it does nothing. (for

example: `try:` and `except`). Python is also slightly different to other programming languages in its `for` loops also have an `else:` clause. Functions are used extensively in programming languages: just as in maths - the function takes an input as an argument, and returns a single result. The functions that we create are called 'user-defined functions'. They a.) allow us to re-use bits of code without having to write multiple times and b.) allow 'abstraction' - easier to understand what a block of code does. Examples for DNA sequencing:

- A function which computes the GC percentage of a sequence
- Checking that a DNa sequence has an in-frame stop codon
- A function to reverse complement a DNA sequence.

The general structure of the function is:

```
def functionname(input args):
        #document the function here
        function code here
        return output
```

`help(functionname)` will return 'string that documents the function'. An important concept: the scope of a variable declaration - the contents within which the program remembers it. A varible can either have a local or a global scope: a global variable is defined outside all functions, and local within functions. Similarly, functions can only see variables which are local (passed) to them. A boolean function tells us if something is true or false and can be used to evaluate conditionals. An example of a function to check for stop codons (note `frame=0` is the default value if no frame value passed to the function):

```
def hasstopcodon(dna,frame=0):
    #this function looks for stop coons
    stopcodonfound=False
    stopcodons=['tga','tag','taa']
    for i in range(frame,len(dna),3):
        codon=dna[i:i+3].lower()
        if codon in stopcodons:
            stopcodonfound=True
            break
        break
```

A reverse complement function is very useful for making the reverse complement of a sequence. For example: `revseq=reversestring(seq), compseq=complement(revseq)`. An example of this, including joins and splits and, importantly, *list comprehension*. One final important thing for this section: a variable number of function arguments. For example: `def myfunction(first,second,third,*therest)`.

Modules are a way to put functions together in a file: python files with a `.py` extension. For example: `import dnautil`, when dnautil is in the PYTHONPATH or your current working directory. To check the path list: `import sys` and `sys.path`. If it doesnt contain the path to the location where your function file is, you can add it: `sys.path.append(path)` where (path) is the path to your functionfile/module. You can import all functions and their defintions using something like `from module import *`. Packages are a way to group modules together into a larger collection. For example: module name A.B designated a submodule named B in a package named A. Each package is a directory, which has to have a special file: __init__.py. This indiciates to python that the directory contains a python package to be imported. We can group modules into one package by putting the package .py files into the same subdirectory as the init file. We can import in two ways: e.g `import package.module`, or `from package import module`.

To read from a file: `open(filename, mode)`, where mode can be 'r' for read (default mode), 'w' for write to the file (overwrite), 'a' for append mode. The result of an `open` function call is a file object. We can use try and excepts to check if a file exists before we try to open it. A simple way to read the file is to iterate over it line by line: `for line in f: print(line)`. Another way is to use `f.read()`. We can go to specific lines of the file using `f.seek()`. We can use `f.write()` to write to a file using the 'w' or 'a' option with a file open. It's good practice to use `f.close()` in order to free up system resources.

To build a dictionary containing all sequences from a FASTA file: open the file, read the line: is the line a header? If yes, get the sequence name and create a new dictionary entry. If not: update sequence in dictionary. Then check whether there are more lines in the file: if not - close the file. We can then retrieve the key-value pairs. The `sys` module allows us to process command line arguments. When we run a script/program in the unix environment, there are standard streams recognized by the program:

- `stdin` - standard in - a stream of data (text) which a program can read. Unless redirected, standard input is expected from the keyboard which started the program.
- `stdout` - standard out is the stream which writes the output data.
- `stderr` - standard error is another output stream used to output error messages (in addition to `stdout` if required)

We can give two different outputs: stdout and stderr from the command line (redirecting two different streams):

```
myprogram | myscript.sh 1>programoutput.txt 2>errormessages.txt
```

Within the unix environment using `stdin`, you need to enter Ctrl+D to end the input. You can also call external programs from inside of Python. We can do this with the `call()` function in the subprocess module. For example: `import subprocess, subprocess.call()`. An genomic example of a subprocess call:

```
subprocess.call(['tophat','genomemouseidx','PEreads1.fq.gz','PEreads2.fq.gz'])
```

BioPython: founded in 1999 - a large collection of modules and scripts for bioinformatics and research. Includes parsers for various bioformatics file formats (such as FASTA, Genbank, etc) with tools for accessing NCBI, etc. Download from http://biopython.org/wiki/Download and `import Bio`. To align a sequence against any other reference sequence, we can use **BLAST** (BioPython has full functionality with the Blast servers): `from Bio.Blast import NCBIWWW` - the web methods from NCBI:

```
from Bio.Blastr import NCBIWWW
fastastring=open('myseq.fa'}.read()
results=NCBIWWW.qblast('blastn','nt',fastastring)
```

There are many parameters to set,although we choose the default ones. Blast may take a few minutes to come back, outputing an .xml file as a BLAST record (e.g. which species it matches what against). For alignment, you get HSP record : high scoring pairs for each separate alignment. Blast limits to 50 alignments. We can write simple scripts to write a threshold for E-values, etc. (i.e. unlikely to be chance matches).