Python for Bioinformatics

by Stuart Brown, NYU Medical School

Contents

Computing Basics

We will use **Python** as a programming language in this class. It has some advantages as a teaching tool and as a first language for the non-programmer. Python is becoming increasingly popular among bioinformaticians.

All programming languages are built from the same basic elements:

- data
- operators
- flow control

We can elaborate a bit more on these

1) Data types
Bioinformatics data is not so different from other types of data used in big computing applications such as physics, business data, web documents, etc. There are simple and more complex 'data structures'. There are three basic data types:

Strings = `'GATCCATGCGAGACCCTTGA'`
Numbers = `7,  123.455,  4.2e-14`
Boolean = `True, False`

More complex data structures allow the integration of basic data types, and also restrict parts of the data (ie. a field or column) to represent a particular type of values, so that software can confidently perform a specific operation on the data.

Variables (a named container for other data types)
Lists: `['a', 'Drosophila melanogaster', 13.456, [1,8,9.89], True]`
Dictionaries:
```
{'ATT' : 'I',
 'CTT' : 'L',
```

```
'GTT' : 'V',
'TTT' : 'F'}
```

2) Operators

Operators include the basic math functions: `+, -, /, *, **` (raise to power)

Comparisons: `>, <, >=, <=, ==`

Boolean operators: `and, or, not`

More complex operators are also known as **functions**. They can deal with file I/O, more complex math, or other manipulations of data. Python has a number of different types of functions with different syntax, some put one or more arguments in parentheses (x,y), others use a dot (.) operator, and others work on an empty argument ().

```
print(x)
open('filename', r)
read.line(filehandle)
my_list.append(42)
write(data, 'filename')
len(string)
exit()
```

3) Flow control

Programs need to make decisions, and have controlled looping (repeat operations for a specific number of times).

Decision operators: `if, esif, else`

Looping operators:

```
for  x  in list:
while  a == True:
```

special flow control: `break, pass`

In Python, users can create their own **functions** or use functions within code written by others (known as modules), which act like subroutines. The function is defined (usually near the start of the program, or in an additional file), then the program sends date to the function using a "function call", the flow of the program moves over to the code of the function, then when the function is completed, the flow returns back to the program with whatever new values (or output) have been produced by the function.

Python navigation tricks

When you are using Python interactively (ie. type `python` in the terminal on phoenix), you get a special cursor symbol (>>>) which indicates that you are no longer in Linux-land. You need to use Python operators to do very basic stuff like change directories

Python is very modular. The basic environment that loads when you type the **python** command (or open the IDLE application, or run a script) has a minimal set of functions. In order to do almost anything useful, you will need to **import** some additional **modules**. The standard install of Python has many of the commonly used modules already stored in a `/bin` directory, you just need to use the `import` command to use them. Other more specialized tools, such as **BioPython** need to be downloaded and installed before you can use them.

You need to import a module called **os** to communicate with the operating system.

```
>>> import os
```

To find your current directory location, use the **getcwd** command from `os`:

```
>>> os.getcwd()
```

To change your current working directory use **chdir**:
[directory names and file names are treated like strings, they need be in quotes]

```
>>> os.chdir ('/Users/stu/Python')
```

To get a list of files in the current directory use **listdir**:

```
>>> os.listdir('.')
```

Create a new directory

```
>>> os.mkdir('My_scripts')
```

Math and Functions

Python can do simple math like a calculator. Type the following expressions into an interactive Python session, hit the enter/return key and observe the results:

```
>>> 2 + 2
>>> 6 -3
>>> 8 / 3
>>> 9*3
>>> 9**3
```

This is a list of built-in Python functions: https://docs.python.org/3/library/functions.html
To do more complex math, you need to import the **`math`** module.

```
>>> import math
```

To use functions from the math module, type **`math`** then a dot (.) then the name of the function in **`math`**. To get a list of all functions in the **`math`** module, type **`help(math)`**. Type the following expressions:

```
>>> math.sqrt(36)
>>> math.log10(30)
>>> math.floor(-1 * math.log10(30))
>>> math.pi
>>> math.pow(2,8)
```

Working with Variables

A variable is a named container for some data object. In Python variables are created with a simple assignment (=) statement such as:

```
x = 1
```

In this statement, the value 1 is assigned to the variable name **x** with the "=" assignment operator. If a different value is assigned to **x**, then it replaces 1. Variable names may contain letters and numbers, but may not start with a number. Variable names are case sensitive. You cannot use symbols like **+** or **-** in the names of variables. The underscore character (_) is often used to make long variable names more readable. The dash character (-) is not allowed in any Python names. Python uses names that start with underline characters for special values, so it is best to avoid them in your own programs.

Once it is created, a variable can be used in place of any value in any expression. For example, variables can be used in a math statement.

```
>>> a = 9
>>> b = 2
>>> sum = a + b
>>> print(sum / 2)
```

A variable can hold many different types of data including strings, lists, dictionaries, filehandles, etc. The data type of a variable does not need to be declared in advance, and it can change during an interactive session or while a program is running. In fact, there are functions that convert between different data types. For example, many functions, such as concatenation only work on strings, so you can convert a number into a string with the **str()** function:

```
x = 128
print ('A is equal to ' + x)   #note the error
print ('A is equal to ' + str(x))
```

When data is entered interactively into a running program from the keyboard it is automatically considered a string (see I/O section below). You can convert a string of number characters into an integer, or a floating point (decimal) number with the **int()** or **float()** functions.

```
x = '2'
y = '3.14159'
print(x + y)
print(int(x) + int(y))              # note the error
print(int(x) + float(y))
```

Bioinformatics applications

We are going to get started using Python with some manipulations of DNA and protein sequences. In fact, there are many different ways to accomplish these tasks in Python. Many sophisticated modules have been developed to make common bioinformatics tasks simple, but it is useful to learn how to control sequence strings with simple Python commands. For all of the examples below, type the commands shown in your own terminal running Python (don't type the >>> prompt) and check that you get the same result as shown.

```
Note: You can install Python on your own Mac, Windows or Linux machine. The Install
directions at Python.org are fairly straightforward:   https://www.python.org/downloads/
In addition to the command line version, you can also install the IDLE graphical interface,
which provides a few nice enhancements that make coding easier. You just open the IDLE or
Python application to get the  >>>  prompt.
```

First, open Python. In your terminal type: **`python`**
You should see the **`>>>`** prompt.

String Methods

Create a variable called DNA with a string of DNA letters (use quotes).

```
>>> DNA = 'GAATTC'
```

Single and double quote marks are basically interchangeable in Python, but you need to be consistent for each quoted string. The equal sign (=) is always used to assign a value to a variable. The value on the right is assigned to the variable name on the left. Spaces within a command line are generally optional, but make the code more readable.

Use the **`print`** command to print the string to the terminal:

```
>>> print(DNA)
GAATTC
```

Ok, now create a second variable name and assign a different DNA string to it. Lets add a poly-A tail.

```
>>> polyA = 'AAAAAA'
```

Now we are going to “ligate” the polyA tail onto the DNA string using the + operator (concatenate) and put the result into a new variable name. In Python, many operators work on strings in fairly logical ways.

```
>>> DNA_aaa = DNA + polyA
>>> print(DNA_aaa)
GAATTCAAAAAA
```

The `print` command can do operations on its own, but note that this does not capture the result in a variable for use later on.

```
>>> print(DNA + polyA)
GAATTCAAAAAA
```

OK, now we are going to “transcribe” the string of DNA letters into RNA by changing all of the ‘T’ letters into ‘U’ (Thymine to Uracil). Python has a nice **`.replace`** method for strings. A method is a function that uses the dot (.) notation like this: `string.replace('old', 'new')`

```
>>> RNA_aaa = DNA_aaa.replace('T','U')
>>> print(RNA_aaa)
GAAUUCAAAAAA
```

The **`.replace(a,b)`** method can also be used to remove newlines from a string (to replace `'\n'` with an empty string `''`)

```
text.replace('\n', '')
```

There are nice string methods to do things like change the case of letters. This is handy if you take data from external files or user input (from the keyboard) where the case may not be known. Python is case sensitive, so a string like `'`**`GATC`**`'` will not match `'`**`gatc`**`'`. Use the method `string.lower()` to change DNAaaa to lower case. Note that `string.lower()` does not take any argument inside the parentheses.

```
>>> DNA_aaa_Low = DNA_aaa.lower()
>>> print DNA_aaa_Low
gaauucaaaaaa
```

String Slicing

DNA exists in the cell as a double stranded molecule where each base is paired with its complement, A-T and G-C. There is also a direction or polarity to the DNA sequence, so that it always reads left to right on the upper strand (5' to 3') and right to left (also 5' to 3') on the complementary strand - see below.

5'-> GAATTCAAAAAA -<3'
3'-> CTTAACTTTTTT -<5'

Many important DNA patterns such as restriction sites and transcription factor binding sites can be found on either strand of the DNA sequence, but in bioinformatics we often do not know which strand we are dealing with. Many algorithms that search for patterns in the sequence need to check both strands by computing the reverse complement. Python does not have a built-in method to reverse strings, but it can be done quite easily using something called string slicing. A string can be sliced using index numbers. Python likes to count things by calling the first item zero. So for a string such as the one in our **DNAaaa** variable, you get the first letter by putting an index number in square brackets, like this:

```
>>> DNA_aaa[0]
'G'
```

You can get a subset of a string by using begin and end value separated by a colon. Very strangely, this string slicing method only takes the number of letters up to, but NOT INCLUDING the last index number.

```
>>> DNA_aaa[1:4]
'AAT'
```

However, if you leave out the begin or end index number, it takes the string from the beginning or the end. The colon with no index numbers get the whole string.

```
>>> DNA_aaa[:4]
'GAAT'
```

```
>>> DNA_aaa[4:]
'TCAAAAAA'
```

The string slicing method can also take characters that are spaced out at intervals by using a 'step' index as a third parameter inside the square brackets.

```
>>> DNA_aaa[1:8:3]
'ATA'
```

A negative step value counts backward from the end of the string. Use an index of -1 to reverse the string. Leaving out both the start and end indexes includes the whole string in the slice.

```
>>> rev_DNA = DNA_aaa[::-1]
print rev_DNA
'AAAAAACTTAAG'
```

For Loops and If Statements

OK, now to create the complement. We need to replace A with T, T with A, G with C, and C with G. You might think that we can just repeat the `string.replace` method, for each of the 4 DNA letters, but this creates a problem. If you replace all A's with T's, then replace all T's with A's, what happens to the string? Try it and see what you get.

```
>>> rev_DNAc1 = rev_DNA.replace('A','T')
>>> rev_DNAc2 = rev_DNA.replace('T','A')
>>> print(rev_DNAc2)
```

???? #Where did all the T's go?

Instead, we need to step through the string one letter at a time, replace each letter with it's complement and then write a new string. Lets deal with the letter replacement function first. We will test which letter is found, then write it's complement to a new string. Python uses a **`for`** loop to step through each character of a string. **`For`** loops do a lot of work with with a very simple syntax. After the **`'for'`** keyword, you must create a new variable that holds each letter as you work through the loop (it is traditional to use the letter **i** as the index variable in a **`for`** loop), then the name of the variable that holds the string, followed by a colon. [*The colon is essential, beginning programmers often forget it.*] After the colon, a new line must be indented (exactly 4 spaces). Then any commands inside the loop are executed. The loop is finished when the indented lines end. The loop repeats automatically for all of the letters in the string, then ends. The syntax is as follows:

```
DNA = 'GATC'
for i in DNA:
    print(i)

G
A
T
C
```

The program needs to make decisions. Specifically, it needs to figure out what DNA base is in the current value of i and change to its complement. The **if** operator is for decision making. **If** uses a syntax similar to **for.** The **if** keyword is followed by a ***conditional*** expression that can be tested as True or False, and then a colon. In this case we will test if **i** is equal to the string 'A'. Note that the double equal symbol must be used to test if two things are equal.

if i == ‘A’:

When the test is true, then the indented statements following the colon are executed. In this case, we will assign the complementary base into another variable called **j** (use a single = sign to assign a value to a variable). When the test is false, the indented statement is skipped and the program moves on. To add a second test, use the keyword **elif** (an abbreviation of else_if), which will only be checked when the first **if** test is false. Finally, if all tests are false, you can add a final **else:** clause, which will only be executed in the case that none of the other conditions are true.

```
if i == 'A':
    j = 'T'
elif i == 'T':
    j = 'A'
elif i == 'C':
    j = 'G'
elif i == 'G':
    j = 'C'
else:
    print(‘letter in string not a DNA base’)
```

To collect the complementary letters, we need to create a new string **comp_DNA** that grows one letter at a time. The new string needs to start out empty (''), then each time through the **for** loop another letter is added from variable **j** into **comp_DNA.** To put the whole thing together, it looks like this:

```
DNA = 'GATC'
comp_DNA = ''
for i in DNA:
    if i == 'A':
        j = 'T'
    elif i == 'T':
        j = 'A'
    elif i == 'C':
    j = 'G'
    elif i == 'G':
        j = 'C'
    else:
        print(‘letter in string not a DNA base’)
    comp_DNA = comp_DNA + j
print comp_DNA
```

Working with Lists

So far we have only looked at variables that hold the value of a single string or number. It would be very inconvenient to create a unique named variable for every single number or text piece of data that a program needs to handle. Python has a data type known as a **list** that can hold any amount of data in many different types. Lists can be created and assigned to a variable name using square brackets and commas to separate items. A list can hold many different numbers:

```
>>> my_list = [9, 2, 4, 78, 98.6, 43.2]
>>> print(my_list)
```

The elements of a list are accessed according to their position in the list, using an index number. Python counts lists using an index that starts at zero.

```
>>> my_list[0]
>>> 9
```

Python has some built-in functions that work on lists of numbers including **sum**, **max**, **min**; and the method **list.sort()**. Many other functions that work on lists are available in the standard set of modules such as **math**, **random**, and **statistics**.

```
>>> sum(my_list)
>>> max(my_list)
>>> my_list.sort()
>>> print(my_list)

>>> import statistics
>>> statistics.mean(my_list)
```

A list can hold strings:

```
>>> my_words = ['apple', 'pear', 'GAATTC', 'Mycobacterium', "large snake"]
>>> print(my_words)
```

You can mix and match numbers and text within a list. Lists can also contain variables and other lists, which can be explicit, or included as variable names.

```
>>> mixed_list = ['apple', 'GAATTC', 22, 3.14159, sum, my_list, ['a','b','c']]
>>> print(mixed_list)
>>> ['apple', 'GAATTC', 22, 3.14159, 9.234, [2, 4, 6, 78, 98.6, 43.2], ['a', 'b', 'c']]
```

A matrix or array can be considered a list of lists, where each sub-list has the same number of elements, so the first list counts the columns, and the sub-lists count the rows.

In addition to direct assignment using the square brackets, elements can be added to an existing list using the **list.append** method. You can only append to an existing list. Sometimes it is necessary to create an empty list (using just a pair of square brackets) early in a program so that you can append things to it later on.

```
>>> test_list = []
>>> test_list.append(99.99)
>>> test_list.append('spam')
>>> print(test_list)
```

It is quite easy to convert back and forth from a string into a list of letters and from a list into a string using the **list()** and **join()** methods. If we start with a DNA sequence as a string and assign it to a variable:

```
>>> seq_string = "TATAAT"
```

Then the **list()** method can break it into a list of single character elements:

```
>>> seq_list = list(seq_string)
>>> print(seq_list)
['T', 'A', 'T', 'A', 'A', 'T']
```

The **join()** method concatenates the elements of a list into a string, but join requires that you specifiy a separator. The separator can be set to an empty string [sep='']. **Join()** is a dot method with a syntax that may seem a bit backwards; the join is acting on the separator with the elements of the list: sep.join(list)

```
>>> sep = ''
>>> new_str = sep.join(seq_list)
>>> print(new_str)
TATAAT
```

Custom Functions

Now that you have a working program for creating the complement of a DNA sequence, it would be nice to combine it with the code for reverse slicing a string, and put the whole thing into a reusable function called **rev_comp** that can be called by your program whenever you need to reverse-complement some DNA. Python uses a simple syntax for users to create custom functions (known as subroutines in other programming languages) that is very similar to the **if** and **for** flow control statements. Functions are created with the **def** keyword, followed by the name of the new function, followed by some parentheses, followed by the ever useful colon. Inside the parentheses are the **arguments** used by the function. Arguments are variable names that receive values that the function uses as its starting data. In the case of **rev_comp**, the function needs an input string of DNA.

The definition of our new rev_comp function looks like this:

```
def rev_comp(DNA):
```

After the definition line, the entire body of the function is indented. When the function completes its work, it could print something to the terminal or write something to a file, but most of the time it returns some newly computed value back to another program. The final line of the function specifies the **return** value.

The complete **rev_comp** function now looks like this

```
>>> def rev_comp(DNA):
        comp_DNA = ''
        j = ''
        for i in DNA:
                if i == 'A':
                        j = 'T'
                elif i == 'T':
                        j = 'A'
                elif i == 'C':
                        j = 'G'
                elif i == 'G':
                        j = 'C'
                else:
                        print('input letter is not a DNA base')
                comp_DNA = comp_DNA + j
        revcomp_DNA = comp_DNA[::-1]
        return revcomp_DNA
```

Now you can test **rev_comp** by calling it as a function in another statement and providing a DNA string as the argument to the function.

```
>>> DNA2 = 'TTGGCGCGTATA'
>>> print(rev_comp(DNA2))
TATACGCGCCAA
```

Working with Files and Input

Saving Python Programs as Executable Files

Now that you have written a longer Python program with a custom defined function, its time to learn how to save your work as a file of executable code that can be reused later or shared with other people. Python programs are saved as plain text files with names that end with the **.py** extension. You can use any text editor to write the code, but it is important that the final version of the file is saved as plain text (not RTF format, which is the default for the Mac TextEdit program, and not as MS Word doc or docx, which is a total mess on the Unix level). It may be best to copy the text of your program into a Linux editor such as emacs or nano, or write it in the IDLE

All Python programs must start with a line that identifies the program to the operating system as a script to be executed with the Pyton interpreter. This line is called the “shebang” line and it usually looks like this:

```
#!/usr/bin/env python
```

The first two characters (**#!**) are called hash-bang or “shebang”. Together they indicate that this file is a script and that it will be executed using the program interpreter located at the path

described on this line. The **/usr/bin/env** command is a generic way of describing the location of the Python program itself. Since the actual Python code is located in different places in different operating systems, the use of **/env** is intended to look up the location of Python according to the path being used on your computer. Sometimes Python programs are written using a hard coded path to **/usr/bin/python** or **/usr/local/bin/python**, but these locations may not work on every system.

Traditionally, the first program students write in a new programming language prints the statement “Hello world!”. So make a ‘Hello world!’ program in Python like this:

```
#!/usr/bin/env python
print(‘Hello world!’)
```

Write your program in a text file, add the shebang line at the top, and save as text with the **.py** extension. There is one more step that is necessary on most computer systems (Mac OSX and all Linux including phoenix). You must change the permissions of the file so that the operating system will allow you to run it as ‘executable’ code. The Linux **chmod u+x** command is used to add execute permissions like this:

```
chmod u+x myscript.py
```

If you want to be fancy, you can assume that all files that end in .py in your current working directory should be made executable, and you can use the * wildcard in place of the filename:

```
chmod u+x *.py
```

On Windows systems with Python (properly) installed, text files named with the **.py** extension are automatically recognized as executable files.

Now to run your Python program. In Linux (and Mac), open a terminal, cd to the directory where your hello.py program has been saved and just type python and the filename at the command line:

```
python hello.py
```

On Windows, open a Command Prompt, cd to the directory where your hello.py program has been saved, and type: `py hello.py`

Hopefully your program will run smoothly, or at least produce some output.

Reading Files

Bioinformatics often deals with large files of data. It does not make sense to type lots of data in by keyboard or to paste it directly into the text part of your programs. Python can read text files stored on your computer using the **open()** and **read()** methods. A file must be assigned to a

variable name (called a "*filehandle*") when it is opened, then it can be read. The filename must be quoted. A second parameter is used to specify if the file is open for reading (**'r'**), writing (**'w'**), or both (**'r+'**).

```
>>> my_file = open('A0PQ23.fasta', 'r')
>>> data = my_file.read()
```

The **read()** method grabs the entire contents of the file. It is often more convenient to use the **readline()** method to grab one line at a time from the file, **strip()** off the newline symbol from the end of the line, and then do something with the line of text, such as save it in a list. It is very common to read all of the lines of a file using a **for** loop, so Python has made a very simple syntax for this task: `for line in file:`

```
my_file = open('A0PQ23.fasta', 'r')
seq_list =[]
for line in my_file:
        my_file.readline()
        line.strip()
        seq_list.append(line)
```

###Note that you need to create the **seq_list** variable as an empty list before you can append data to it.###

In some cases, such as searching for a pattern within a DNA or protein sequence, it may be better to join all the lines together into one long string so that you don't have to worry about patterns that wrap across lines. The **join()** method concatenates the elements of a list, but it has a strange syntax, since it expects to use some type of separator between the elements. To concatenate elements directly, assign the separator to an empty string `('')`.

```
>>> codon_list = ['GAT', 'CCC', 'ATG', 'GGG']
>>> sep = ''
>>> sep.join(codon_list)with
'GATCCCATGGGG'
```

Bioinformatics data often comes in tab delimited structure, such as gene name, chromosome location, gene expression intensity, or fold change. There are some special functions for specific data formats (see Biopython, discussed below), but any tab delimited (or comma delimited) data file can be imported by the **csv** module. The **csv.reader** function reads a tab delimited file [https://docs.python.org/2/library/csv.html]. In the example shown below, the syntax of the open statement is changed to use the with … as x: format, which creates a loop for reading the data file, only reads the lines of the file until they have all been read, and provides a check that the file exists.

```
import csv
with open('yourfile.csv') as f:

        reader = csv.reader(f, delimiter='\t')
```

```
        data = list(reader)
print data[0][2]
```

Note that this imports the elements as strings. If you want to treat them as numbers, simply convert to an **int** or **float** when you use the element:

```
print(float(d[0][2]) / float(d[1][2]))
```

One very simple and useful thing that you can do in Python with lists of data is to find intersections and differences between lists. We often do gene expression experiments and end up with some lists of Differentially Expressed (DE) genes. Perhaps we have a Control, experimental condition A, condition B, and condition C. We calculate DE between each of the 3 experimental conditions vs the Control using a sophisticated method (SAM for microarray, Cufflinks or edgeR for RNA-seq), with the resulting output stored in tab delimited files. Now we want to know which genes are DE in condition A, but not in B, and which are DE in both A and C. If we import the data files (dataA, dataB, dataC) into Python using the **csv.reader** function, then the lists of gene names should be in column[0] of each file (assuming the gene names were in the first column of the original tab delimited text files).

```
a = dataA[0]
b= dataB[0]
c= dataC[0]

difference = list(set(a) - set(b))

both = list(set(a) and set(c))
```