Beginner Python Programming for Biology

Allison E. Mann, June 14, 2018

What is Python?

Python is a flexible easy to read programming language often used in the biological sciences. It was designed in the late 1980s by Guido van Rossum (Python's official Benevolent Dictator for Life) and named after the British skit group, Monty Python. Unlike other common programming languages (e.g., Perl) python relies heavily on whitespace to structure its code. In this tutorial the basic syntax of python is introduced including how to write simple functions, and loops. In addition, two helpful python modules – Pandas and BioPython – as well as the interactive IPython, are introduced.

Installing Python

- Windows: https://www.python.org/downloads/windows/
- If you have Mac or Linux OS, python should already be installed on your computer. Type the following into the command line to see which version is installed:

  python --version
- If you receive an error message or no version name is displayed, you may need to install the latest version of python here: https://www.python.org/downloads/mac-osx/ (Mac OS).

NOTE: This tutorial runs on python version 3 or above. If you have python version 2.7 or below installed, there will be some syntax issues.

Installing Pandas:

- Linux:

```
sudo apt-get install python-pip
sudo pip install numpy
```

```
sudo pip install pandas
```

- Mac OS: https://pandas.pydata.org/pandas-docs/stable/install.html
- Windows: https://pandas.pydata.org/pandas-docs/stable/install.html

Installing BioPython:

- http://biopython.org/DIST/docs/install/Installation.html#htoc3

Installing IPython:

- https://ipython.org/install.html

More Python resources:

- Learn Python the Hard Way: https://learnpythonthehardway.org/
- BioPython Tutorial and Cookbook: http://biopython.org/DIST/docs/tutorial/Tutorial.html
- Code Academy (Python): https://www.codecademy.com/learn/learn-python

Python Syntax:

Python programs are written as text documents that end with the .py extension. Typically, python scripts begin with #!/usr/bin/python (#! is called a shebang), which indicates the location of your python install and makes the script executable (though this isn't strictly necessary). Unlike other programming languages (e.g., perl or R), blocks of code are not specified by brackets or bracers. Instead, blocks of code in python are indicated by indentation and colons. For example:

```
if (x > y):
     print(“x is greater than y”)
else:
     print(“x is less than or equal to y”)
```

In this script, a print statement is run depending on which criteria is met as set up in the if and else statements. It's good practice to include comments in your code with further information on what a particular block of code or statement does (or is intended to do). Comments in python are indicated by the # sign. For example:

```
if (x > y):
    print(“x is greater than y”)
else:
    print(“x is less than or equal to y”) #if x is not greater than y this statement will be read
```

Anything beyond the # symbol on the same line will be ignored by the python interpretor.

Data Types in Python:

Variables or objects that you define or import into the python workspace have different data types. Data types are not declared by the user (as is the case with lower level languages like C), instead they are inferred from the assigned statement. It's important to note that because of this, sometimes python will infer the **WRONG** data type. Be sure to check that your variables are assigned in the proper format to avoid problems down the road.Arial

The most common data types in python include:

- **Boolean:** True/False
- **Integer:** Any full number (e.g., 5)
- **Float:** A number with a fractional value (e.g., 1.234)
- **String:** A letter, sentence, word, etc.

Statements and Expressions:

Variables or objects are assigned to some value with the = sign. For example, if I wanted this string: “Hello world” to be assigned to a variable I could do so by:

```
myString = “Hello world”
```

This stores “Hello world” to the variable name myString but does nothing else to it. If I wanted my string to be printed to the

screen after its assignment, you need to call the print statement:

```
print(myString)
```

Math and other Operators:

Math operators:
- Addition: +
- Subtraction: -
- Multiplication *
- Division /
- Exponentiation **
- Modulus %

Comparison operators:
- Equal to ==
- Not equal to !=
- More, less than > <

Important Python Data Types:

List – a mutable (or changeable) and ordered sequence of elements – delineated by square brackets [ ]

Dictionary – an object type that consists of keys that are associated with particular values. Think of a word in a dictionary as they key and the associated definition as the value. Delineated by curly brackets and a colon {key:value}

Tuple – an immutable (unchangeable) sequence of objects (very similar to a list) – delineated by parentheses ( )

Importing Libraries:

May modules or expressions in python are not included by default and instead must be loaded into the python environment (or added to your python script). For example, say you would like to use the BioPython library in your script. You can do so by adding the following line to the beginning of your code:

```
import Bio
```

Where import is the library importation command and Bio is the name of the BioPython library. More typically, however, you'll

want to include specific tools from the larger library. For example, if I wanted to import SeqIO, a function that can be used to import sequence files, from BioPython I could do so like so:

```
from Bio import SeqIO
```

It's usually a good idea to import specific modules from your library instead of the full library to cut down on memory usage. Additionally, you may need to install the library before it is available to use. Installation instructions for BioPython can be found here: http://biopython.org/DIST/docs/install/Installation.html

Writing a Function:

Python functions are a good way of setting up small modules that you can link together to create larger programs. The basic structure of a python function is below.

```
def myFunctionName(optional input parameters):

     some set of conditions to run
```

After writing your function you need to call it within the script somewhere after it is defined.

```
myFunctionName(input)
```

Example – My First Python Program:

Open the file called helloWorld.py for viewing.

```
#!/usr/bin/python3

#import statements

import sys

#doc string explaining what/how to use the program

'''Usage: python helloWorld.py <your name>'''

#a simple function
```

```
def main():
     if len(sys.argv) >= 2:
          name = sys.argv[1]
     else:
          print("Please enter your name:")
          name = input()
     print('Hello', name)
#standard way of calling the main function
if __name__ == '__main__':
     main()
```

Python in the Interactive Shell (IPython):

One of the nicest things about python is it's interactive shell environment IPython. You will need to install this separately but it provides an easy way to debug or test out parts of your script. It also has the extra benefit of tab completion and you can use most Unix command lines within the shell. Instructions for downloading and installing ipython can be found here: http://ipython.org/install.html

Example – Parsing a Genbank File in IPython:

So let's test this out by writing a small script that converts a genbank formatted file to a fasta file. First open up a terminal and run IPython by typing:

```
ipython
```

Now write out your script by first importing the SeqIO module from BioPython

```
from Bio import SeqIO
```

In one statement you can now read in your genbank file and parse through it line by line to extract information from it in a loop.

```
for seqRecord in SeqIO.parse("my_file.gb", "genbank"):
    print(seqRecord)
```

The results of this command shows you all of the information in the genbank file for your sequence record object. Note, you can call this any variable name you want – think of it as **for each element in my file: do this**. There are a total of six individual genbank records in this file, each of them have different features and annotations associated with them. Let's look at some of the most common features: the sequence **ID** (aka the NCBI accession number and version), the sequence **Name** (NCBI accession), the actual gene **Sequence** itsArialelf, and the sequence **Description** (Usually the organism, locus, and other important information about the sequence itself).

Try running the above command but instead of seqRecord by itself, change the command to one of the features below.

```
seqRecord.id

seqRecord.name

seqRecord.description

seqRecord.seq
```

Features of a genbank file (like the ones above) are a required part of the format but there is other information that we can pull from this file. So for example, what if we wanted to only list the organism from which each of our genbank records is generated from?

```
#E.g., the following is part of the output of print(seqRecord)

/organism=Brassica napus

/taxonomy=['Eukaryota', 'Viridiplantae', 'Embryophyta', 'Tracheophyta', 'Spermatophyta',
'Magnoliophyta', 'eudicotyledons', 'core eudicots', 'Rosidae', 'eurosids II', 'Brassicales',
'Brassicaceae',

'Brassica']
```

These are annotations to the sequence record, to pull these we need to indicate that the annotation is an aspect of our sequence record with the . operator as well as select those annotations that are listed as “organism” with square brackets.

```
for seqRecord in SeqIO.parse("my_file.gb", "genbank"):
    print(seqRecord.annotations["organism"])
```

Now that we've taken a look at our genbank file, we decide we want to convert the sequence data in this file to a workable fasta format. We can also do this with BioPython!

First you need to define an output handle (think temporary place holder for your eventual output file)

```
output_handle = open("my_file.fa", "w")
```

Now we can loop through each record in our genbank file and write the sequence feature of each in the fasta format written to our output handle.

```
for seqRecord in SeqIO.parse("my_file.gb", "genbank"):
    output_handle.write(">%s %s\n%s\n" % (seqRecord.id, seqRecord.description,
seqRecord.seq))
```

Example – Calculating the Percent Identity of Two Fasta Formatted Sequences:

Open the file percent_identity.py for viewing

```
import sys #library that allows you to communicate via command line

from Bio import AlignIO #load AlignIO (input/output) class from BioPython library

"""Problem: You have an alignment file of two 16S rRNA genes (fasta format) and want to
calculate the percent identity between them"""

#add fancy colors to the output of your screen (in this case green when the script is
complete)

class colors:
```

```
    COMPLETE = '\033[92m'
#initialize our data/variables
align = AlignIO.read(sys.argv[1], "fasta")
first_seq = list(align[0])
next_seq = list(align[1])
def pid():
    matches = 0 #initialize values for loop
    for nucleotide in range(0, len(first_seq)):
        if first_seq[nucleotide] == next_seq[nucleotide]:
                matches += 1
    perc_id = (matches*100)/float((len(first_seq)))
    print(colors.COMPLETE + ("Percent identity: %.2f" % perc_id))
pid()
```

Example – Data Munging:

Open the file otu_table_fix.py for viewing.

```
import sys #library that allows you to communicate via command line
import pandas as pd #load and set up alias for pandas library
"""PROBLEM: you have a tab delimited OTU table called filtered_otu_table_L6.txt with redundant
taxa strings that need to be collapsed"""
data = pd.read_csv(sys.argv[1], sep="\t", skiprows=1)
```

```
def fixOTU():
     fixed_data = data.groupby("#OTU ID").sum()
     #how many duplicate records were in the file?
     duplicates = data.shape[0] - fixed_data.shape[0]
     print("Number of duplicate rows to be collapsed: %i" %duplicates)
     #now write your fixed OTU table to file
     with open("collapsed_otu_table.txt", "w") as outfile:
          fixed_data.to_csv(outfile, sep="\t")
     outfile.close() #you're done writing to file, close it (not necessary but good practice)
     print("Complete, collapsed OTU table written to: collapsed_otu_table.txt") #tell the user
that it's done!
fixOTU()
```