Lab 7

Importing and Creating Functions

Today we will learn about how to import and use functions in Python. A function is a block of code that performs a specific action, and can be referenced multiple times. Functions are important because they allow you to reuse pieces of code with different inputs and parameters. Python has many built-in and preloaded functions (e.g. `len()`, `range()`, `sum()`, `min()`, and `max()`); however, there are also many other functions that are stored separately in libraries, which are not automatically loaded. In addition, it is possible to write user-defined functions that can be embedded within a script, or stored in separate user-created libraries. You will learn how to import functions that are not automatically loaded by Python. You will also learn how to write your own functions and store and call them within a script.

You will follow along in a Jupyter notebook, and run the code as we go through it together. Feel free to try to play around as we progress to get a better feel for how things work.

REMEMBER, THE INTERNET IS YOUR FRIEND: Python.org Tutorials

Part A: Importing Functions

Python automatically loads some built-in functions for users, e.g. `len()`, `range()`, `sum()`, `min()`, and `max()`. However, there are also many other functions and objects that are stored in libraries that Python does not load automatically. These have many uses, such as interacting with the operating system or the user (e.g. `sys`, `os`); string operations (e.g. `re`); advanced computation (e.g. `math`, `random`); and accessing new data types (e.g. `datetime`, `sets`). These are included in the default Python installation and do not have to be separately downloaded as packages and installed.

To access these functions and objects, we need to import the library. We use the `import` function to load new libraries that are not loaded by default. This is typically done immediately after `#!/usr/bin/python`. The library can be renamed using import ... as ..., and functions and objects in the library can be accessed using a dot (.). Let's try importing and using some functions.

Follow the directions below and enter the missing information. When you are ready to run the code, hit "`CTRL+ENTER`" and jupyter will run your code for you:

In [1]:
```
#!/usr/bin/python

#importing libraries
import math
import random
```

```
import glob, sys #import multiple libraries on one line
import re as regex #import and rename a library

#example of built-in functions
#note that we did not import them!
number_list = [1,2,3,4,5]

#print the sum
print ('sum:', sum(number_list))
#get the length of the list with the len() functino
print ('len:',len(number_list))

#example of imported functions
#access objects and functions in a library with a dot (.)
five_cubed = math.pow(5,3)
print ('5 to the 3 power:', five_cubed)
#get the square root of the number 10 with the sqrt() function
square_root_10 = math.sqrt(10)
print ('Square root of 10:', round(square_root_10, 2))
#note that round is a built-in function!
```

```
sum: 15
len: 5
5 to the 3 power: 125.0
Square root of 10: 3.16
```

Part B: Using `random`

Now that we've learned how to import and access specific functions, let's work with functions in the `random` library. This library implements various functions for generating pseudo-random numbers or sequences by uniformly selecting from a range of numbers or elements, by generating a random permutation of a list, or by random sampling items from a list without replacement.

In [2]:
```
#randomly generate a number between 0 and 1
print ('random number between 0 and 1: ', random.random())

#randomly generate an integer between (a) and (b)
#format: random.randint(a,b)
print ('random integer between 1 and 100: ', random.randint(1, 100))

#randomly rolling a die
#format: randrange(start, stop, step)
#the value of STOP is +1 your desired stopping point
print ('random die roll: ', random.randrange(1,7))
```

```
random number between 0 and 1:  0.40878040887933176
random integer between 1 and 100:  30
random die roll:  6
```

Note: because the values are being generated randomly, it is unlikely that your output will look the same every time. Try running the code multiple times – you should see the values changing!

You can see now how `random` can be incredibly useful for making decisions, e.g. picking a number randomly for the lottery, rolling a die for a game, or choosing between "heads" or

"tails" for a coin toss!

Part C: using `random` to create a random DNA sequence

Now that we've looked at some basic functions in `random`, let's use the function `random.choice()` to randomly choose an element from a list of elements. We can use this to build a DNA sequence by adding one random nucleotide at a time.

In [7]:
```
#randomly flipping a coin
#format: random.choice(list) - literally a list of choices!
print ('random coin toss: ', random.choice(['heads', 'tails']))

#use the example above to print a random nucleotide
print ('random nucleotide: ', random.choice(['A','C','T','G']))

#let's build a random DNA sequence of length 10
#initialize a variable to store your sequence
sequence = ''

#use range() and for to have 10 turns (length 10)
for i in range(10):
    #add a randomly generated nucleotide to your sequence
    sequence = sequence + random.choice(['A','C','T','G'])

#print the random DNA sequence
print ('random 10-mer: ', sequence)
```

```
random coin toss:  tails
random nucleotide:  G
random 10-mer:  CGAAGTTCCG
```

Try running your script multiple times!

Part D: creating your own functions

Now we know how to import libraries and functions, but what if there is no pre-existing package that does what we want? We can write our own functions! Writing functions let's you organize your code into discrete "blocks" which makes reading and running code easier. Also, if you have to do the same thing over and over again, with different inputs or parameters, it is useful to write a function. For example, if we want to generate 5 random DNA sequences, instead of copying and pasting our code 5 times, we could just write function and then call it 5 times.

Functions are defined with `def`, followed by a name for the function (e.g. `generate_random_DNA`), any parameters in parentheses, and finally a colon (:). All code associated with the function is indented below `def`. Later, the function can be called by its name, plus any necessary parameters. Functions typically have some kind of output, which the user can access by using `return`.

In [8]:
```
print ('Using our own functions:')

#create a function
```

```
#IMPORTANT: all code associated with the function is INDENTED!
#LENGTH is a parameter that the user will supply later in the code
def generate_random_DNA(length):
    #initialize a variable to store the sequence
    sequence = ''
    #make the number of turns equal to the the length of the sequence
    for i in range(length):
        #add a randomly generated nucleotide to the end
        sequence = sequence + random.choice(['A', 'T', 'C', 'G'])
    #return the result to the user
    return sequence

#now let's use our function!
#using a function is called CALLING the function
#we will call our function 5 times, to make 5 random sequences
for i in range(5):
    print ('random 10-mer: ', generate_random_DNA(10))

#we can also save the output of generate_random_DNA(10)
rand_DNA = generate_random_DNA(10)
#then we can change it using other functions, or print it
print ("Saved sequence: ", rand_DNA)
rand_DNA_last5 = rand_DNA[-5:]
print ("Last five bases in rand_DNA: ", rand_DNA_last5)
```

```
Using our own functions:
random 10-mer:  CTAGGAAAGT
random 10-mer:  CAGGTACACC
random 10-mer:  TGCCATGCGG
random 10-mer:  TGCGAAAGTG
random 10-mer:  GATAAAACTC
Saved sequence:  CCGACTAAGC
Last five bases in rand_DNA:  TAAGC
```

Lab Task

Using **Part C** of this lab, you have already become familiar with using `random` to generate a random DNA sequence; in **Part D**, you learned how to incorporate this into your own function that can be called repeatedly. For today's lab task, you'll use these concepts to generate a biased DNA sequence, and then compare it to a randomly generated DNA sequence. You will make use of `random`, write your own functions, and import pre-existing functions, as well.

Here are some functions that you will need. Be sure to run this block of code before you start work on your lab task:

In [17]:
```
import math

#This function returns the average and standard deviation of a list of numbers
def avg_std(numbers):
    N = len(numbers)
    total = sum(numbers)
    avg = float(total) / N
    stdev = 0.0
    for x in numbers:
```

```
        stdev = stdev + (x - avg)**2
    stdev = math.sqrt(stdev / N)
    return "%.2f +/- %.2f" % (avg, stdev)

#This function returns a list of frequencies of Adenines in DNA sequences
def freq_A(sequence):
    num_A = sequence.count('A')
    len_seq = len(sequence)
    frequency = (float(num_A) / len_seq)
    return frequency
```

OBJECTIVES:

In the Python cell below:

1. Using `random`, create your own function that generates random DNA sequences, exactly like we did in **Part D** of the lab. The length of the sequence should be **variable**. **Hint**: don't forget to `import random`!
2. Write a new function, called `generate_biased_DNA`, using `random` to generate a DNA sequence with twice as many As than C/T/G. The length of the sequence should be variable. **Hint**: There are many ways to do this. Ultimately your goal is to have a sequence that is approximately 50% As, and the other 50% is made equally of C/T/G. **IMPORTANT**: make sure your functions are defined before calling them. It is good practice to write functions at the start of your code, not in the middle.
3. Use a `for` loop to generate a random sequence of **length 20**, followed by a biased sequence of **length 20**. Make the `for` loop run for **10 turns**.
4. Now we will make use of the functions `avg_std()` and `freq_A()` defined above. Within the `for` loop that you wrote in step 3, use the function `freq_A()` to count the frequency of adenines (As) in the random and biased DNA sequences. During each iteration, save the output of `freq_A()` for each DNA type. **Hint**: Declare lists outside the `for` loop, then use `.append()` to add the numbers to the respective lists; make sure to have one list for random DNAs, and one list for biased DNAs.
5. Finally, use the function `avg_std()` to compute the average and standard deviation for each of your lists. The average frequency of adenines in the biased DNA set should be higher than in the random DNA set. Is this what you observe? Do you ever observe the numbers being similar enough that the biased DNA set looks random?
6. **Don't forget to write comments!**

#An example output for part 3 may look like: # : random 10-mer biased 10-mer 0 : CCGTTAACCCAATGTACACG GGCTTTAAACACAGTAGAAT 1 : GATTACCAGTACTGACTTGT GGAATGTACAGACAACTAGC 2 : CGCTGCCGGGTGGCGAGAAA TAGACGCATAAGAAACTAGA 3 : TCTTATCTCGTCTGCAATCG TCAAACCATAGAAAAAACGC 4 : AATCGAAACTCATAACGCAA AATGTGAATGTTTCAAGTAA 5 : AGAATGGCAACGCACGTCAA ATAACATAACCGACTCAGGT 6 : CACGACGGGCAGTTGTAAGT ACTGCGGTCTAAAGTGAAAA 7 : CAAAAGCCAGTAAGGGGCGC AGGCGACCACGTCGGTATGA 8 : TATCCCGGGTTGCGCTGGAA CATGGAAAAAGACACAAACA 9 : AACGACTGTGGTCATCTGTT AACTAAATTCATAAGGAATG

In [27]:
```
#write your python code here
import random

#create a function to generate random DNA sequences
```

```
#the length of the sequence should be variable
def generate_random_DNA(length):
    #initialize a variable to store the sequence
    sequence = ''
    #make the number of turns equal to the the length of the sequence
    for i in range(length):
        #add a randomly generated nucleotide to the end
        sequence = sequence + random.choice(['A', 'T', 'C', 'G'])
    #return the result to the user
    return sequence

#create a new function to generate a biased DNA sequence with twice as many As
def generate_biased_DNA(length):
    # store all nucleotides in a list
    nucleotides = ['A', 'T', 'C', 'G']
    # initialize a new variable to store the biased sequence
    biased_DNA = ''
    #add a randomly generated nucleotide to the end with weight
    biased_DNA = biased_DNA.join(random.choices(nucleotides, cum_weights =(50,6
    #return the result
    return biased_DNA

#generate a random sequence follow by a biased sequence for 10 times
print("# : random 10-mer biased 10-mer")
#initialize a variable to store the frequency of As
freq_As_random =[]
freq_As_biased =[]
for n in range(10):
    print(n,":",generate_random_DNA(20), generate_biased_DNA(20))
    #use function freq_A() to countthe frequency of adenines (As)
    #in the random and biased DNA sequences
    freq_As_random.append(freq_A(generate_random_DNA(20)))
    freq_As_biased.append(freq_A(generate_biased_DNA(20)))

#show the frequency of As in each sequence to the user
print("frequency of adenines (As) in the random DNA sequence",freq_As_random, '
      "frequency of adenines (As) in the biased DNA sequence",freq_As_biased)

#compute the average and standard deviation for frequency of adenines in ramdor
avg_std(numbers)
```

```
# : random 10-mer biased 10-mer
0 : AGGAAAAGATTAGCGCAGGC AAATGACAACTAACATACGA
1 : AGTGTCGAACCTAGACTATC CACGCTGAAGGAAAAGGAAA
2 : AGGGGCTAGATCCAATACTT TGAAAAAGTAGCATCGGAAA
3 : TTAAGATCGACGGGATGAAG CGATGCTAAAAACGATCAGA
4 : TACGACTACGGCTGGGGCAT TCCTAAAGGAATGAAAGTAA
5 : CCTTGTTTTGTGGCGGGAGA AATAACGACACGGAGGCAAA
6 : CCTAACAGTTGGCGTACCAA ATCAATTAAGCTAGTTGGGA
7 : CTTTCATCTGCCTAGCAACA TAATGAAAAGGGTACAAATA
8 : GTTGACCATTTGCGGATAGC GAGCAAGATCGAGGACAGAA
9 : CCTCGGTCCACTAGCTACGA GTGCTAGAATCTATGTAAGG
frequency of adenines (As) in the random DNA sequence [0.4, 0.25, 0.2, 0.3, 0.
2, 0.15, 0.1, 0.2, 0.25, 0.35]
 frequency of adenines (As) in the biased DNA sequence [0.5, 0.65, 0.35, 0.55,
0.55, 0.6, 0.6, 0.55, 0.45, 0.55]
```