**functions**

A fairly frequent task in programming is to do a task many times. For example, maybe you need to process two files and calculate the number of lines in each. It makes sense to write the code only one time. In addition, we can edit the code in one place.

For example, you may have a dna sequence in a file that you want to know the proportion of each nucleotide.

dna\_obj=open("dna")  
seq=dna\_obj.read()  
seq=seq.rstrip("\n")  
dna\_obj.close()  
  
length=len(seq)  
a\_s=seq.count("A")  
g\_s=seq.count("G")  
c\_s=seq.count("C")  
t\_s=seq.count("T")  
print ("The frequencies of A , G , C, and T are " + str(a\_s/length) + " " + str(g\_s/length) + " " + str(c\_s/length) + " " + str(t\_s/length))

If we want to calculate nucleotide frequencies for many sequences, we make a function. We create a block of code and give it a name using the def keyword.

For example, the simplest function prints a set of error messages.

def print\_error():  
 print ("That was an absolutely dreadful choice")  
 exit ("I refuse to continue. Goodbye!")  
  
from sys import argv  
print(argv[1])  
if (int(argv[1])<5):  
 print\_error()  
else:  
 print ("Goodbye")

Here is a function that calculates nucleotide frequencies.

def get\_freqs(seq):  
 seq=seq.rstrip("\n")  
 length=len(seq)  
 a\_s=seq.count("A")  
 g\_s=seq.count("G")  
 c\_s=seq.count("C")  
 t\_s=seq.count("T")  
 return [a\_s,g\_s,c\_s,t\_s]  
  
freqs=get\_freqs("AGTC")  
print ("The list is ")  
print(freqs)

Or we could return a dictionary

def get\_freqs(seq):  
 seq=seq.rstrip("\n")  
 length=len(seq)  
 dict={'A' : seq.count("A"), 'G' : seq.count("G"), 'C' : seq.count("C"), 'T' : seq.count("T")}  
 return dict  
  
freqs=get\_freqs("AGTC")  
print ("The list is ")  
for key in freqs.keys():  
 print (key)  
 print (freqs.get(key))

Note that variables we create inside a function cannot be accessed on the outside. For example,

print ("The length is " + length)

print ("Goodbye")

#adding this code throws an error:  
# print ("The length is " + str(length))  
#NameError: name 'length' is not defined

We can calculate the frequencies instead by using this code

dict={'A' : seq.count("A")/length, 'G' : seq.count("G")/length, 'C' : seq.count("C")/length, 'T' : seq.count("T")/length}

Division may produce a large number of numbers after the decimal place. The round() function takes the number we want to round and the number of significant figures.

print (round(freqs.get(key),3))

Functions become more useful if you can pass data to them. Items to be passed should be between parentheses. We can pass more than one argument to a function and the arguments can be different data types.

For example:

###

def f\_colour(fruit):  
 dict={'apples' : 'red or yellow',  
 'grapes' : 'red or green'}  
 return(dict['apples'])

print(f\_colour ("apples"))

####

def add\_them\_up (nums):  
 return(sum(nums))  
  
print(add\_them\_up([3,4,5]))

###

def name\_print (first,middle,last):  
 print (first + middle + last)  
 return()  
  
name\_print('First','Middle','Last')

### We can also return multiple values. e.g.

def add\_them\_up(nums):  
 return (sum(nums), len(nums))  
  
print(add\_them\_up([3, 4, 5]))

Note that you can use a variable in the main part of the program in the function without passing the variable to the function, but this practice is discouraged. You should make the variables given to the function clear and be able to use the function potentially with different data.

### Function arguments can have defaults.

def p\_num (number, sig\_figs=2):  
 print(round(number,sig\_figs))  
p\_num(23.234324)  
p\_num(23.234324,4)

**classes**

So far, we have called methods on objects such as strings and lists. We can make our own types of objects and our own methods. An object is an instance of a class. A class is like a blueprint for an object.

var\_name= open("file.txt")

For example, we would say that var\_name refers to an object that is an instance of the file class. By defining a class, objects are not created directly. The class instead describes what an instance of a class will be like. We can create as many instances as we like.

One can take different functions and use them to build up a class. Let’s start with functions. Here are two

def get\_AT(my\_dna):  
 length=len(my\_dna)  
 a\_count=my\_dna.count("A")  
 t\_count= my\_dna.count("T")  
 at\_count=(a\_count + t\_count)/length  
 return at\_count  
  
def complement(my\_dna):  
 replacement1=my\_dna.replace("A","t")  
 replacement2= replacement1.replace("T","a")  
 replacement3= replacement2.replace("G","c")  
 replacement4= replacement3.replace("C","g")  
 return replacement4.upper()  
  
seq="TATAGGCCATACG"  
print(get\_AT(seq)) *#at content*print(complement(seq)) *#complement*

This script has two functions and works well. The script is useful, but we may also want to add the species name and the gene name to the code to keep track of the sequences. We want to attach a bit of metadata to the sequence.

seq="TATAGGCCATACG"

species="A. thaliana"

gene="Adh"  
print (species + " " + gene + " " + seq)

print(get\_AT(seq)) *#at content*print(complement(seq)) *#complement*

This solution works for one sequence, but we can’t type in species name, sequence, and gene every time. We need to link the metadata with the output. One way is through a data structure would work. For example, a dictionary may have a sequence records. The key may be a sequence ID. The value would be an array with three items corresponding to its species, gene name, and sequence. This would work well, but creating a class that represents a DNA sequence is even better. Instances of this class can be generated and used throughout the program.

What instance variables and methods will the class have? Instance variables are variables that belong to an object. Methods apply to the object. We will create a class DNARecord with three instance variables and two methods.

class DNARecord (object):  
 sequence="ACGTTATTGATTT" *#variables of the class* gene\_name= "ADH"  
 species= "A. thaliana"  
  
 def complement(self): *#the first method* replacement1=self.sequence.replace("A","t")  
 replacement2= replacement1.replace("T","a")  
 replacement3= replacement2.replace("G","c")  
 replacement4= replacement3.replace("C","g")  
 return replacement4.upper()  
  
 def get\_AT(self): *#the second method* length=len(self.sequence)  
 a\_count=self.sequence.count("A")  
 t\_count= self.sequence.count("T")  
 at\_count=(a\_count + t\_count)/length  
 return at\_count  
  
d= DNARecord() *# a new instance of the DNA Record Class*print ("Created a record for " + d.gene\_name + " from " + d.species) *#accessing class variable*print ("AT is " + str(d.get\_AT())) *#calling method*print ('Complement is ' + d.complement())

Note that we access d’s attributes using a “variable\_name.attribute\_name” nomenclature. The “self” term above is how we refer to the object inside the method. To refer to the DNA sequence of the record, we use self.sequence. Whenever we make a method call on our object, python sets the value of the self variable.

A limitation of the class is that the sequence, species, and gene\_name are set as part of the class definition. We do not want every instance of the class to have these values, and we can change them after creating an object. We simply set them after the object has been created.

d1=DNARecord()

d1.sequence= "AAAATTTT"

d1.gene\_name="TGA"

d1.species="Zea mays"

d2=DNARecord()

d2.sequence= "AAAACTAATAG"

d2.gene\_name="TB1"

d2.species="Zea mays"

for r in [d1, d2]:  
 print (r.gene\_name + " from " + r.species)  
 print (str(r.get\_AT()) + "is AT frequency for d1")  
 print ("… and complement is …")  
 print (r.complement())

We can make entering values easier by adding a new method to the class.

class DNARecord (object):

#sequence="ACGTTATTGATTT" #variables of the class

#gene\_name= "ADH"

#species= "A. thaliana"

def complement(self): #the first method

replacement1=self.sequence.replace("A","t")

replacement2= replacement1.replace("T","a")

replacement3= replacement2.replace("G","c")

replacement4= replacement3.replace("C","g")

return replacement4.upper()

def get\_AT(self): #the second method

length=len(self.sequence)

a\_count=self.sequence.count("A")

t\_count= self.sequence.count("T")

at\_count=(a\_count + t\_count)/length

return at\_count

**def set\_variables(self, new\_seq, new\_gene\_name, new\_species):**

**self.sequence = new\_seq**

**self.gene\_name = new\_gene\_name**

**self.species = new\_species**

**d1=DNARecord()**

**d1.set\_variables("AAAATTTT","TGA","Zea mays")**

**d2=DNARecord()**

**d2.set\_variables("AAAACTAATAG","TB1","Zea mays")**

print ("hello")

for r in [d1, d2]:

print (r.gene\_name + " from " + r.species)

print (str(r.get\_AT()) + "is AT frequency for d1")

print ("… and complement is …")

print (r.complement())

Now that we have set the variables, there is no reason to keep the assignments we had previously. We do need to remember to set that variables for an object after we create it. If we try to use an object before its variables are set, we get an error.

d3=DNARecord()

print(d3.complement()) #AttributeError: 'DNARecord' object has no attribute 'sequence'