**Example script to interact with the filesystem**

Counting the number of k-length sequence patterns amongst a group could be useful for identifying highly represented sequences like transcription factor binding sites amongst a group of promoter sequences. Let’s do this. First, we can make a function that captures kmers. Then we can read in sequences from files and process them (from Advanced Python for Biologists)

def make\_kmer (seq,kmer\_size):  
 kmers=[]  
 for start in range(0, len(seq)-kmer\_size+1, 1):  
 kmer=seq[start: start+kmer\_size]  
 kmers.append(kmer)  
 return kmers  
  
*#a="AGTAGAGAGATAGATATAGAG"  
#print(make\_kmer(a,5))*import os  
kmer\_counts= {} *#define a dict to count kmer numbers*for file\_name in os.listdir("./Sequences"):  
 if file\_name.endswith(".txt"):  
 print ("sequence detected: " + file\_name)  
 dna\_file= open ("./Sequences/"+file\_name)  
 for line in dna\_file:  
 dna=line.rstrip("\n")  
 for kmer in make\_kmer(dna,5): *#args are the sequence and kmer length* count=kmer\_counts.get(kmer,0)  
 kmer\_counts[kmer]=count+1  
  
print (kmer\_counts) #prints the kmers and counts

**Additional data structures**

Tuples

Tuples (pronounced “couples”) are similar to lists. Many of the list methods work for tuples, but list methods that change the elements within the tuple will generate an error. Tuples are immutable, e.g. unchangeable. They are usually used for small collections of often disparate data that are tightly coupled. Instead of brackets (‘[]’), tuples are defined with parentheses ().

t=(4,5,6)

print(type(t))

print (str(t[0]))

#t[0]=6 #generates error

It may be beneficial to keep values immutable, and the data can be faster to process. For example, one may have a list of genes with unique positions (as in the .gff file). For example, I pretend below we have gene start sites on a chromosome and depict the direction of transcription. These data make good tuples.

g1=("atg1", "+", 1000)  
g2=("atg2", "-", 3000)  
g3=("atg3", "+", 5000)  
print (g1[1])

Lists of lists and numpy arrays

Lists can contain strings, numbers, and objects. For example here is a list of file objects

list1=[open("seq1.txt"),open("seq2.txt"),open("seq3.txt")]

Lists can contain other lists

list2=[[1,2,3],[4,5,6],[7,8,9]]

This list is like a matrix

list2=[[1,2,3],

[4,5,6],

[7,8,9]]

print(list2[0]) #is [1,2,3]

print(list2[0][2]) #is 3

#getting a column is a bit more tricky. The numerical python (numpy) package allows easier array processing.

Lists of dicts and lists of tuples

Here is a list of dicts that could be used to store data.

records= [

{ 'name' : 'atg1', 'strand' : '+', 'length' : 1000 },

{ 'name' : 'atg2', 'strand' : '-', 'length' : 3000 },

{ 'name' : 'atg3', 'strand' : '+', 'length' : 5000 }

]

The dicts created here are different than in previous examples. In previous dicts, we had a key value pair, and all of the values were of a similar type. For example, a super-hero name key would call up an alias value. Here, the values are of different types, string and integer. Also, the keys, although unique within each dictionary, are being used as labels for the values rather than unique identifiers. Finally, each dict is anonymous.

for record in records:

print ("Gene name: " + record['name'])

print ("Gene length: " + str(record['length']))

This gene data is the kind of data that tuples are good for. We can make a list of tuples.

records=[

('atg1', '+', 1000),

('atg2', '-', 3000),

('atg3', '+', 5000)

]

Now, we use the location in the tuple to recover data

for record in records:

print ("Gene name: " + record[0])

print ("Gene length: " + str(record[2]))

#Or

for record in records:

(name,strand,length)=record #unpacking the tuple

print ("Gene name: " + name)

print ("Gene length: " + str(length))

Dicts of tuples

We can see that the list of tuples above is good for iterating over the data but not so good for pulling out specific data. We would have to loop through records to identify a specific gene, an inefficient thing to do. If we have unique identifiers and want to look up a record, it makes more sense to make a dict of tuples.

records= {

'atg1' : ('+', 1000),

'atg2': ('-', 3000),

'atg3': ('+',5000 )

}

my\_record=records['atg3']

my\_record=records.get('atg3')  
print(my\_record[1]) #returns 5000

Or unpack the tuple at the same time.

(strand, length)= records.get('atg3')

print(length)

Dicts of sets:

Recall that a set is an unordered collection of immutable values. It can be created with {1,2,3} or by passing any sequential object to the set constructor, like set([1,2,3]). Sets are handy for membership testing. We can have a dict of sets that contain differentially expressed genes.

genes = {

'cold': {'A','B','C'},

'drought':{'A','D','E','F'},

'heat':{'A','B','G','H'}

}

print('A' in genes['drought']) #or “not in”

Below is a “for key, value in dict.items():” command that will investigate the genes in each condition

for condition, genes in genes.items():

if 'A' in genes:

print (condition)

print (type(genes['drought'])

##or we could test the proportion overlap

print(genes['drought'].union(genes['heat']))

print(genes['drought'].intersection(genes['heat']))

Dicts of lists:

We can use lists as values for dicts. Going back to the kmer count script, let’s say we want to keep track of where in a sequence the kmer is found. We can have each unique kmer act as a key to a list that has kmer start positions.

kmer\_dict\_list={}  
  
dna\_file= open ("./Sequences/seq\_2.txt")  
dna\_init= dna\_file.read()  
dna=dna\_init.rstrip("\n")  
  
for start in range(0, len(dna)-6, 1):  
 kmer=dna[start: start+5] *#kmer size is five* pos\_list=kmer\_dict\_list.get(kmer,[])  
 pos\_list.append(start)  
 kmer\_dict\_list[kmer]=pos\_list  
  
print (kmer\_dict\_list)