Introduction to Programming - Python DNA project final.

Thor Skjold Haagensen.221190.Thhaa16 Section H9.
Supervisor: Peter Schneider-Kamp
DM550

1. november 2016

Indhold

1 Specification

By choosing the dna project, I've had to make a program that can read a .txt file, make everything uppercase and delete the letter "N"which we, for the purpose of the assignment weren't interested in. Also the program has to find TATAA boxes and from that find start codons and end codons, finally storing them in a list.
Afterwards the program has to have a class for amino acids which has a function for reading another file containing names of amino acids and the codons that makes them. Then distinguish the names from the codons. For each acid there has to be created an object of the class with the attributes corresponding to the abbreviation the short name and the longname and the object has to contain a list of the codons.
After that, the program has to have a Ribosome class which takes an acid object and adds mappings from the names to the codons which it does with a dictionary, second it has to invert that dictionary, making a search with codons available
Lastly the program has a Protean class with an associated function that can take a string of codons and return the corresponding amino acids. That function has a perimeter mode which returns the acids as either their abbreviations, short names or long names.
The class also has a bonus function that can test all the strings found in task 2 - 4 and return the correct acids.

2 Design

In part two of the project i wrote a code that could open the ChrX file, make everything uppercase letters, delete the N's and newlines / whitespaces and lastly it had to be able to print the stripped text. I decided to do this locally in a function that way I could solve task 3 and 4 within the same function for task 3 I used a while loop with the find() method to find TATAAA boxes but instead of storing them I found that if I had a index that increased each time I found a TATAAA box, I would be able to iterate over the whole file finding the corresponding ATG and from that finding the stop codons. Once my program found all 3 closest stop codons from the ATG, the program would then proceed to compared the length of the 3 strings and the shortest would then be the one i was interested in, and also the one which would be appended to my string index which I stored globally so that I would be able to use the information later.

in task 7 I created an Acid class which contains attributes for the abbreviation, short name, long name and a list of codons that each acid is translated to. The program takes advantage of that the start of the amino acids names are indicated by a »"and the corresponding codons didn't. Then the program iterates over the dna-codons file with a for loop and each time I got to a new line with the »"symbol it would split the line at the ","that separated the abbreviation, short name and longname and then use these as attributes for a object of the Acid class, and every line in the document until »"will then be added to that objects list of codons.
For task 8 the program has a Ribosome class which has mappings from the acids

to the codons, to this end the program uses three dictionaries
(one for each of the names), with the acids as keys and the codons as corresponding values, making a search by acid names available. The program then inverts the three dictionaries and stores them in three new dictionaries, enabling search by codons. For task 9 The program has a Protean class that inherits the dictionaries and functions from the Ribosome class, The program also has a function that can take a string of codons and return the corresponding amino acids. In solving this problem the program takes a parameter, either 1, 2 or 3. if 1 is given it searches the inverted dictionary where the values are the abbreviations and so on. The function also takes a string of codons. making this part i had to make some decisions on how the program reads the strings, the reason is that if the strings are not divisible by 3 there would be some excess characters which would make the program return an error when searching the dictionary. therefore the program has a rule that the codon needs to be longer than 2 when searching. When a string is not divisible by 3 it means that some of the codons in the string uses some of the end codon to make an acid, and the rest of the codon is then less than 3. In the cases where the string is divisible by 3 the end codons would appear in their full form eg. TGA, TAG and TAA. This would also give the program an error because of searching for something that is not in the dictionary, therefore the program will cut the last character of the string with a string slice and the string then falls under the >2 rule.

3 Implementation

Task 2 was solved Using a variable that opens a file containing codons then using 3 replace statements the read() and upper() and then printing the variable.

```
1 chrx = open("/Users/Thor/Desktop/chrX.fa").read().upper().replace("N","")
      .replace("\n","").replace(" ","")
2
3 print(chrx)
```

When planning task 3 and 4 i figured i might as well have the open command in the same function.

```
1 def findatg():
2     open("/Users/Thor/Desktop/chrX.fa").read().upper().replace("N","").
          replace("\n","").replace(" ","")
3     index_count = 0
4     new_index = 0
5     while new_index != -1:
6         new_index = chrx.find("TATAAA",index_count)
7         atg = chrx.find("ATG",index_count+21,index_count+36)
8         index_count = new_index + 1
```

In the task of finding the TATAA boxes i took advantage that the find() function returns -1 if the searched element is not found in the string. Therefore im using a while loop that runs until there are no more TATAA boxes in the string, the while loop also keeps track of where it is in the file using the index count which is the the number of the character where the TATAA was found.

When the program finds a TATAA box it then uses the find() command to look for the sequence ATG, this sequence had to be found 15 - 30 characters after the TATAA box. Therefore im searching the index count of the "TATAAA"+ 6 for the box itself and then + 15 to +15 more.

```
strgindex = []

def findatg():
    chrx = open("/Users/Thor/Desktop/chrX.fa").read().upper().replace("N"
        ,"").replace("\n","").replace(" ","")
    index_count = 0
    new_index = 0
    while new_index != -1:
        new_index = chrx.find("TATAAA",index_count)
        atg = chrx.find("ATG",index_count+21,index_count+36)
        index_count = new_index + 1
        if atg != -1:
            TAG_stop = chrx.find("TAG",atg)
            TAA_stop = chrx.find("TAA",atg)
            TGA_stop = chrx.find("TGA",atg)
            if TAG_stop < TAA_stop and TAG_stop < TGA_stop and TAG_stop >
                atg:
                a = chrx[atg:TAG_stop+3]
                if len(a) > 3:
                    strgindex.append(a)
            elif TAA_stop < TGA_stop and TAA_stop < TAG_stop and TAA_stop
                > atg:
                a = chrx[atg:TAA_stop+3]
                if len(a) > 3:
                    strgindex.append(a)
            elif TGA_stop < TAA_stop and TGA_stop < TAG_stop and TGA_stop
                > atg:
                a = chrx[atg:TGA_stop+3]
                if len(a) > 3:
                strgindex.append(a)

findatg()
```

The final program is displayed above. When the program finds an ATG sequence that satisfy the search criteria, it immediately searches and saves each "TGA", "TAA"and "TAG"in a variable for each eg. "TAG stop". The program then compares the length of each strings and if the shortest string is longer than 5 it will append it to a list called strgindex. the reason the string has to be longer is to avoid strings of the type ATGA, which i interpreted as an unwanted combination of ATG and the stop codon TGA. i do however allow strings of the form ATGTAG, which is just methane with a stop codon, as a quick note, the reason for having the strgindex outside of the function, is that i'm then able to use all the strings for later testing but they could just as well have been local.

For task i've created a class (Acid) that when initialized with a abbreviation, short name and a long name creates an object with those three as attributes and a list called aminolist.

To initialize the class i've created a function called acid creater. The function opens the file dna-codons.cdl. the reason i chose to open it with the command"with open() as"was so i could iterate over the file without the readline() command. After opening the file the program checks whether the next line starts with »"or not. If the start of the line is »"it then removes the »"and splits the line between each ","making a temporary list. it then invokes the Acid class with the 3 elements of the list as its names eg. the attributes. All this is added to a global list called acidlist.
In the case that the lines do not start with »". I use the fact that the codons making the acid comes after the acid names, therefor the program adds every line to the the previously made object's acid list.
The object also has a string method but that is only for testing.

```
1  acidlist = []
2
3  class Acid:
4      """ Represents an Amino acid """
5      def __init__(self,abbreviation,shortname,longname):
6          self.abbreviation = abbreviation
7          self.shortname = shortname
8          self.longname = longname
9          self.aminolist = []
10
11     def __str__(self):
12         return '%s %s %s' %(self.abbreviation,self.shortname,self.
               longname)
13
14
15
16 def acid_creater():
17     with open("/users/thor/desktop/dna-codons.cdl") as fin:
18          for line in fin:
19             if line[0] != ">":
20                  b = line.strip()
21                  if len(b) > 0:
22                      acid.aminolist.append(line.strip())
23                  else:
24                      a = line.replace(">","").split(",")
25                      acid = Acid(a[0],a[1],a[2])
26                      acidlist.append(acid)
27 acid_creater()
```

In task 8 i had to make another class called Ribosome. This class when initialized goes through the acidlist and creates three dictionaries, one for the abbreviation one for the short name and one long name (The one for the abbreviation is name "d abbr"). the dictionary contains the acid names as a keys and it's corresponding aminolist as values. After the three dictionaries are created, the init method then inverts them and adds the result to three other dictionaries (The one for the abbreviation is called d abbr inv).
I'm using the defaultdict() method because I couldn't make a regular dictionary work with lists as keys, this would become an issue when inverting the dictionaries. Again the print function was made for testing.

```
1  class Ribosome:
2  """ stores mappings from amino acids to codons in dictionaries and then
       reverses them in new dictionaries """
3
4      def __init__(self):
5          self.d_abbr = dict()
6          self.d_short = dict()
7          self.d_long = dict()
8
9          self.d_abbr_inv = dict()
10         self.d_short_inv = dict()
11         self.d_long_inv = dict()
12
13         for i in acidlist:
14              self.d_abbr[i.abbreviation.strip()] = (i.aminolist)
15              self.d_short[i.shortname.strip()] = (i.aminolist)
16              self.d_long[i.longname.strip()] = (i.aminolist)
17
18         for k in self.d_abbr:
19             for v in self.d_abbr[k]:
20                 self.d_abbr_inv.setdefault(v,k)
21         for k in self.d_short:
22             for v in self.d_short[k]:
23                 self.d_short_inv.setdefault(v,k)
24         for k in self.d_long:
25             for v in self.d_long[k]:
26                 self.d_long_inv.setdefault(v,k)
27
28                 def printer(self,a):
29                """Able to search the inverted dicts with three or more
                       codon """
30                print(self.d_abbr_inv[a])
31                print(self.d_short_inv[a])
32                print(self.d_long_inv[a])
```

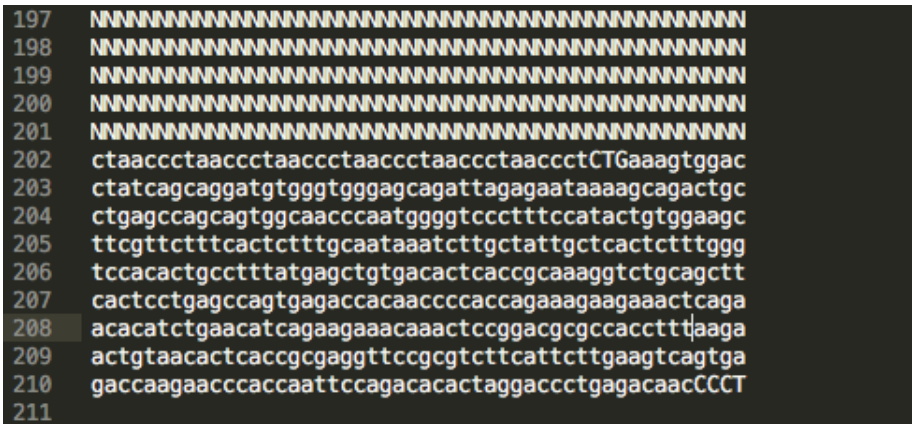
I chose to use inheritance for the last part of the program, that way when Protean is called Ribosome is being called as well triggering the dictionaries in it's init method.
The protean class has a function called translator(self,parameter,codon)that uses the self method referring to itself and the Ribosome class (Because of inheritance), aside from that it takes two arguments, a parameter which can be 1,2 or 3. for an example if the parameter chosen is 3 the function will return the translations in the long name eg. for the codon "TTT"the return will be Phenylalanine. the last argument is codon which takes a string. The string is then (on line 120) split into 3 seperate strings (lists) with list comprehension and saved as a variable "a"the program then searches the chosen dictionary string by string with a for loop, and for each string returns the corresponding values. If the string is a string divisible by 3 and not an accepted codon, the program will give an error and stop. However you can for an example write "ok"and it will return None because of the rule on line 122 that says that the string has to be longer than 2. This was implemented because sometimes there are excess codons in the codon strings which in our interpretation then "burrows"from the

end codon to make up an acid The last function is a codon tester, which runs the translator against all the strings collected in task 2-5 in the strgindex. This function cuts the last letter of the codon to dodge the problem just described.

```
1
2  class Protean(Ribosome):
3
4      def __init__(self):
5      Ribosome.__init__(self)
6
7
8      def translator(self,parameter,codon):
9          """parameter can be 1 for abbreviation, 2 for the short name or 3
               for the long name"""
10
11         if parameter == 1:
12             dict = self.d_abbr_inv
13         elif parameter == 2:
14             dict = self.d_short_inv
15         elif parameter == 3:
16             dict = self.d_long_inv
17         n = 3
18         a = [line[i:i+n] for i in range(0, len(line), n)]
19         for i in a:
20             if len(i) > 2:
21                 print(dict[i])
22             else:
23                 continue
24
25     def codon_tester(self,parameter):
26         for i in strgindex:
27         print(i)
28         print(protean.translator(parameter,i))
29
30
31
32 protean = Protean()
```

4 Testing

For test one we'll be looking at the output of program in task 2, and comparing it to the original text. to this end I'll be using a test document containing the first 210 lines from the original file.



Figur 1: Original text after N's

```
>CHRXCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTCTGAAAGTGGACCTATCAG
CAGGATGTGGGTGGGAGCAGATTAGAGAATAAAAGCAGACTGCCTGAGCCAGCAGTGGCAAC
CCAATGGGGTCCCTTTCCATACTGTGGAAGCTTCGTTCTTTCACTCTTTGCAATAAATCTTG
CTATTGCTCACTCTTTGGGTCCACACTGCCTTTATGAGCTGTGACACTCACCGCAAAGGTCT
GCAGCTTCACTCCTGAGCCAGTGAGACCACAACCCCACCAGAAAGAAGAAACTCAGAACACA
TCTGAACATCAGAAGAAACAAACTCCGGACGCGCCACCTTTAAGAACTGTAACACTCACCGC
GAGGTTCCGCGTCTTCATTCTTGAAGTCAGTGAGACCAAGAACCCACCAATTCCAGACACAC
TAGGACCCTGAGACAACCCCT
```

Figur 2: Programs output

We'll be ignoring the >CHRX it is the start of the document men it doesn't matter for any of the programs functionality. Aside from that we see that all the codon data is exactly the same now only in uppercase letters and that all N's have been removed.

In figure 3 we see the different output for the 3 different parameters. This is also a case where the string is not divisible by 3. If we look at the last 5 letters we see that the base pairs "CA"burrows the "T"from the stop codon "TAA"and therefore the program does not return a lookup error.

```
>>> len("ATGCGTTTTATACATATACCAATGTTATACCATAA")
35

>>> protean.translator(1,"ATGCGTTTTATACATATACCAATGTTATACCATAA")
M
R
F
I
H
I
P
M
L
Y
H

>>> protean.translator(2,"ATGCGTTTTATACATATACCAATGTTATACCATAA")
Met
Arg
Phe
Ile
His
Ile
Pro
Met
Leu
Tyr
His

>>> protean.translator(3,"ATGCGTTTTATACATATACCAATGTTATACCATAA")
Methionine
Arginine
Phenylalanine
Isoleucine
Histidine
Isoleucine
Proline
Methionine
Leucine
Tyrosine
Histidine
```

Figur 3: Searching with parameters

In figure 4 we're testing what we can do to trigger the error, in these cases it's searching for the end codons TAA and TAG.

```
>>> protean.translator(3,"ATGTTTTAG")
Methionine
Phenylalanine
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/Users/Thor/Dna proj backup.py", line 113, in translator
    print(dict[i])
KeyError: 'TAG'

>>> protean.translator(3,"ATGTTTCAT")
Methionine
Phenylalanine
Histidine

>>> protean.translator(3,"ATGTTTCATTGT")
Methionine
Phenylalanine
Histidine
Cysteine

>>> protean.translator(3,"ATGTTTCATTGTTAA")
Methionine
Phenylalanine
Histidine
Cysteine
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/Users/Thor/Dna proj backup.py", line 113, in translator
    print(dict[i])
KeyError: 'TAA'
```

Figur 4: With and without end codons

5 Conclusion

The program reliably translates from codons to acids and could with very little modification translate from acids to codons as well. The error that occurs when searching for an codon sequence that is not in the dictionary, can be seen as positive, it is nice to get a response from the program that the thing you are searching for is in fact not translated to a codon. There are some minor things that could have been done better with the way the program outputs the acids but overall I'm satisfied with what was achieved.

6 Appendix (source code)

```
1
2  strgindex = []
3
4  def findatg():
5      chrx = open("/Users/Thor/Desktop/chrX.fa").read().upper().replace("N"
           ,"").replace("\n","").replace(" ","")
6      index_count = 0
7      new_index = 0
8      while new_index != -1:
9          new_index = chrx.find("TATAAA",index_count)
10         atg = chrx.find("ATG",index_count+21,index_count+36)
11         index_count = new_index + 1
12         if atg != -1:
13             TAG_stop = chrx.find("TAG",atg)
14             TAA_stop = chrx.find("TAA",atg)
15             TGA_stop = chrx.find("TGA",atg)
16             if TAG_stop < TAA_stop and TAG_stop < TGA_stop and TAG_stop >
                   atg:
17                 a = chrx[atg:TAG_stop+3]
18                 if len(a) > 5:
19                     strgindex.append(a)
20             elif TAA_stop < TGA_stop and TAA_stop < TAG_stop and TAA_stop
                   > atg:
21                 a = chrx[atg:TAA_stop+3]
22                 if len(a) > 5:
23                     strgindex.append(a)
24             elif TGA_stop < TAA_stop and TGA_stop < TAG_stop and TGA_stop
                   > atg:
25                 a = chrx[atg:TGA_stop+3]
26                 if len(a) > 5:
27                     strgindex.append(a)
28
29 findatg()
30
31 acidlist = []
32
33 class Acid:
34     """ Represents an Amino acid """
35     def __init__(self,abbreviation,shortname,longname):
36         self.abbreviation = abbreviation
37         self.shortname = shortname
38         self.longname = longname
39         self.aminolist = []
40
41     def __str__(self):
42         return '%s %s %s' %(self.abbreviation,self.shortname,self.
               longname)
43
44 def acid_creater():
45     with open("/users/thor/desktop/dna-codons.cdl") as fin:
46         for line in fin:
```

```
47            if line[0] != ">":
48                b = line.strip()
49                if len(b) > 0:
50                    acid.aminolist.append(line.strip())
51            else:
52                a = line.replace(">","").split(",")
53                acid = Acid(a[0],a[1],a[2])
54                acidlist.append(acid)
55 acid_creater()
56
57
58 class Ribosome:
59     """ stores mappings from amino acids to codons in dictionaries and
          then reverses them in new dictionaries """
60
61     def __init__(self):
62         self.d_abbr = dict()
63         self.d_short = dict()
64         self.d_long = dict()
65
66         self.d_abbr_inv = dict()
67         self.d_short_inv = dict()
68         self.d_long_inv = dict()
69
70         for i in acidlist:
71             self.d_abbr[i.abbreviation.strip()] = (i.aminolist)
72             self.d_short[i.shortname.strip()] = (i.aminolist)
73             self.d_long[i.longname.strip()] = (i.aminolist)
74
75         for k in self.d_abbr:
76             for v in self.d_abbr[k]:
77                 self.d_abbr_inv.setdefault(v,k)
78         for k in self.d_short:
79             for v in self.d_short[k]:
80                 self.d_short_inv.setdefault(v,k)
81         for k in self.d_long:
82             for v in self.d_long[k]:
83                 self.d_long_inv.setdefault(v,k)
84
85     def printer(self,a):
86         """Able to search the inverted dicts with three or more codon"""
87         print(self.d_abbr_inv[a])
88         print(self.d_short_inv[a])
89         print(self.d_long_inv[a])
90
91
92
93 class Protean(Ribosome):
94
95     def __init__(self):
96         Ribosome.__init__(self)
97
98
99     def translator(self,parameter,codon):
```

```
100         #parameter can be 1 for abbreviation, 2 for the short name or 3
                for the long name
101         if parameter == 1:
102             dict = self.d_abbr_inv
103         elif parameter == 2:
104             dict = self.d_short_inv
105         elif parameter == 3:
106             dict = self.d_long_inv
107         line = codon
108         n = 3
109         a = [line[i:i+n] for i in range(0, len(line), n)]
110         for i in a:
111             if len(i) > 2:
112                 print(dict[i])
113             else:
114                 continue
115 
116     def codon_tester(self,parameter):
117         for i in strgindex:
118             print(i)
119             i = i[:-1]
120             print(i)
121             print(protean.translator(parameter,i))
122 
123 
124 
125 protean = Protean()
```