Computational Biology
Tufts University Bio 35
Fall 2021

Assignment 2: Identifying genetic patterns using `Python`

Functions, conditionals, loops, and finding genes

This assignment has two sections. Part one is an ungraded set of computational problems that are intended to help us develop computational skills. Second, we have a set of problems focused on identifying pathogenicity factors in genetic sequences. Both sets of problems are based on source material derived from our textbook, Computing For Biologists. You can find more examples and the original questions at https://www.cs.hmc.edu/twiki/bin/view/CFB/Part1.

1 Ungraded computational exercises

1.1 Functions

Write functions for each of the following. Include a docstring in each function that explains what the function takes as input and what it returns. **Functions are first discussed on page 26 (section 1.2) of CFB – please refer to this section for some information about functions**.

1. A function called `power(x)` that takes a number `x` as input and returns x^x. For example, `power(2)` returns `4` and `power(3)` returns `27`. The exponentiation operator is `**` as in `2 ** 3` will return `8`. (As an experiment, try to find the smallest integer value for `x` such that `power(x)` returns a number greater than 1 million.)

   for this first question we will provide a solution. Subsequent problems will be for you to work out.

   Python function example

   ```
   def power(x):
       return x**x
   ```

2. A function called `stringMultiply(myString, number)` that takes as input a string called `myString` and a positive integer called number and returns a new string that is the concatenation of number copies of `myString`. For example, `stringMultiply('hi', 3)` returns `'hihihi'`.

3. A function called `listMaker(myString, number)` that takes as input a string called `myString` and a positive integer called `number` and returns a list that contains number copies of that `myString`. For example, `listMaker('hi', 3)` returns `['hi', 'hi', 'hi']`.

4. Write a function called `countCodons(DNA)` that takes as input a DNA string (you may assume its length is a multiple of 3) and returns the number of codons in that string. For example, `countCodons("CATGGG")` returns 2.

5. Write a function called `palindromeMaker(input)` that takes a string named `input` and returns a new string that is the input string followed by its reversal. It should work for strings of any length. You may wish to use the built-in `len` function that returns the length of its string. Here's an example of the function in action:

   `palindromeMaker` in action

   ```
   >>> palindromeMaker("good")
   'gooddoog'
   >>> palindromeMaker("ab")
   'abba'
   >>> palindromeMaker("radar")
   'radarradar'
   ```

1.2 Conditionals

In these exercises, you'll practice your `if`. `ifelse`, and else skills, which are also known as "conditionals", because your program will only execute the code if a certain condition is met. Conditionals are reviewed in sections 1.7 and 1.8 of CFB (beginning on page 32).

1. The absolute value of a number is the "positive version" of that number. For example, the absolute value of -42 is 42 and the absolute value of 42 is also 42. Write a function called `absolute(n)` that takes a number `n` as input and returns the absolute value of that number.

2. A palindrome is a string that is the same forwards as backwards. For example `"radar"`, `"wow"`, and `"abba"` are all palindromes. Write a function called `palindrome4(input)` that takes a string of length exactly 4 as input and returns the Boolean `True` if that string is a palindrome and returns the Boolean `False` otherwise. If the string has length less than or greater than 4, the function should also return `False`. Here's an example of the function in action:

palindrome4 in action

```
>>> palindrome4("abba")
True
>>> palindrome4("good")
False
>>> palindrome4("radar")
False
```

3. A DNA sequence that begins with `'ATG'`, ends with `'TGA'`, `'TAG'`, or `'TAA'`, and whose length is a multiple of 3, is called an open reading frame or ORF for short. ORFs are interesting because they can encode proteins. Write a function called `ORF(DNA)` that takes a string called DNA as input and returns the Boolean `True` if that string satisfies the three required conditions. Here's an example of the function in action:

ORF in action

```
>>> ORF('ATGAAATAG')
True
>>> ORF('ATGATAG')
False
>>> ORF('TAATAA')
False
```

   In the second example, the string's length is 7 which is not a multiple of 3. In the third example, the string does not begin with 'ATG'.

4. Write a function called `ORFadviser(DNA)`, which is a modified version of `ORF` that takes a string called `DNA` as input and works as follows:

   - The function returns the string `'This is an ORF'` if the input string satisfies all three of the conditions required of ORFs.
   - Otherwise, if the first three symbols are not 'ATG', the function returns the string `'The first three bases are not 'ATG''`.
   - Otherwise, if the string does not end with `'TGA'`, `'TAG'`, or `'TAA'`, the function returns the string `'The last three bases are not a stop codon'`.
   - Otherwise, the function returns the string `'The string is not of the correct length'`.

1.3 `For` loops

Lists of objects can be traversed with what is known as a `for` loop. In many programming applications we need to go through each item in a list and perform some action on each of the items – this task can be completed with a `for` loop. Loops are discussed in CFB 1.10 and 1.11, beginning on page 38.

1. Write a function called `count(letter, string)` that takes a `letter` as input (a string with just one symbol in it) and returns the number of times that the given letter appears in the given `string`. Here's an example of the function in action:

`count` in action

```
>>> count('b', 'biology terms with z')
   1
   >>> count('z', 'zyzzyva')
   3
   >>> count('z', 'zyzzyzus')
   4
```

2. Imagine that we have a list of DNA sequences and we wish to know how many of them are of a specific length. Write a function called `countLength(DNAlist, length)` that takes a list of DNA strings called `DNAlist` and a positive integer `length` as input and returns the number of strings in the list that have the specified length. Use the built-in `len` function and for loops. Here's an example of the function in action:

`countLength` in action

```
>>> countLength(["ATA", "ATCG", "TTT", "A"], 3)
2
>>> countLength(["AACC", "A", "T"], 2)
0
```

3. Next, write a function called `getLength(DNAlist, length)` that is analogous to the `countLength(DNAlist, length)` but rather than counting the number of strings of the given length, this new function should return a list of the strings of that length. Here's an example of the function in action:

`getLength` in action

```
>>> getLength(["ATA", "ATCG", "TTT", "A"], 3)
['ATA', 'TTT']
>>> getLength(["AACC", "A", "T"], 2)
[]
```

4. The factorial function is defined as follows: For a positive integer `n`, `n` factorial (written `n!`) is the product `n * (n-1) * (n-2) * ... * 1`. For example, `3!` is `3 * 2 * 1 = 6`. Write a function called `factorial(n)` that takes a positive integer `n` as input and returns `n!`. To do this, use the range function and for loops. Here's an example of the function in action:

`factorial` in action

```
>>> factorial(1)
1
>>> factorial(3)
6
>>> factorial(4)
24
```

2 Graded exercise

2.1 Processing DNA sequences

1. **Computing GC Content**: *Place your code in a file called* `gcContent.py` *inside the scripts directory.*

   Recall that bacteria such as *Salmonella* are pathogenic because they have certain genes that allow them to make you sick. Many such genes can be found in clusters called pathogenicity islands in the genome. For researchers studying a pathogenic bacterium, finding these islands is an important step in understanding how the bacterium makes people sick.

Calculating GC content can be useful in helping find pathogenicity islands. The GC content is the proportion of bases that are G's or C's. It turns out that the GC content in pathogenicity islands frequently differs from other regions of the genome. In this problem, you will write a short function to calculate the GC content of a DNA string. You will then use this function on two regions of *Salmonella* DNA, one that comes from a pathogenicity island, and one that does not.

First, download the folder for this week's assignment insert hyperlink and navigate to the `scripts` folder, where you should find a file called `twoSalDNAs.py`. Then at the top of your `Python` file (`gcContent.py`), include the following line:

gcContent.py

```
from twoSalDNAs import *
```

This will import two DNA strings into your environment, `inIsland` and `outsideIsland`. These correspond to one region that is in *Salmonella* pathogenicity island 1, and another region that is not.

In the text, we discussed several functions for calculating GC content. The examples we used made the assumption that the input DNA was of a certain fixed length. Now you should write a better GC content function that can calculate the GC content of a string of any length using a for loop. Call your function `gcContent(DNA)`.

gcContent in action

```
>>> gcContent("ACCGC")
0.8
>>> gcContent("ATACTAAA")
0.125
```

Using this function, calculate the GC content of `inIsland` and `outsideIsland`, and include these values as a comment at the top of your file.

2. **GC content windows**: *Place the code for this exercise in a file called* `gcwindows.py`.

In this problem, we will once again examine GC content in the *Salmonella* genome. You previously wrote a function named `gcContent(DNA)` that computed the GC content of its input string. Now, we'll consider a more realistic model. In particular, you'll write a function that identifies a likely pathogenicity island by exploiting the fact that a pathogenicity island typically has GC content that differs from that of the larger DNA sequence in which it resides.

Assume that we have one long piece of genomic DNA that contains a single pathogenicity island somewhere within it. That DNA sequence can be "chunked" into pieces and the GC content of each piece can be computed separately. By seeing which pieces have relatively high or low GC content, you'll be able to infer roughly where the pathogenicity island resides.

Make sure you are in the scripts directory for the homework, where you should find a file called `salGenomicRegion.py`.

Next, at the top of your file, include the line:

gcwindows.py

```
from salGenomicRegion import *
```

This will import a DNA string called `salDNA` into your environment. This string comes from a larger region surrounding *Salmonella* pathogenicity island 1.

Your task is to write a function called `gcWin(DNA,stepLen)` that takes as input a DNA string and does the following: for each piece of DNA of length `stepLen`, the GC content is calculated (using your existing `gcContent` function from the previous problem) and that value is printed (using `print` rather than `return`).

For example, imagine that our DNA string has length 13 and the `stepLen` is 5. Then the first region is the region `DNA[0:5]`, the next one is `DNA[5:10]`, and the last one is `DNA[10:15]`. Notice that the last slice may

not be of length `stepLen` because of the length of the DNA sequence, but that's fine. Recall that if the string has length 13 then `DNA[10:15]` will simply provide a short slice.

We can use a for loop to go over the DNA such that after each iteration of our loop we skip forward `stepLen` bases. The following syntax of the built-in (i.e., you don't have to write this function, it exists in `Python` already)`range` function will be useful here:

Python's range function

```
>>> range(0,100,10)
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Note how the last number we gave the range function told it how big a step to use.

You can loop over the DNA in this way, slicing out each region of size `stepLen` in turn. For each slice, calculate the GC content by calling `gcContent(DNA)`, and print the result to the screen.

`gcWin` in action

```
>>> gcWin('ATCCGAGGTC',2)
0.0
1.0
0.5
1.0
0.5
```

One you have written this function, run it on `salDNA` with `stepLen` set to 10,000.

Based on the results, where do you think the pathogenicity island begins and ends in this region? Include your answer as a comment at the top of your file.

3. **Loading FASTA files**: *Your code for this problem will be in a file named* `load.py`.

   Throughout this course, we'll be studying a number of computational methods for analyzing biological sequences (e.g., DNA sequences and protein sequences). An important step is obtaining and loading such data into `Python`. That's the primary objective of this problem.

   Obtaining *Salmonella* sequence from NCBI

   First, obtain a DNA sequence from the National Center for Biotechnology Information (NCBI). This is the main organization in the US government that collects and maintains DNA sequence information.

   Go to the NCBI home page (https://www.ncbi.nlm.nih.gov/), and paste the accession number U81861 into the main search box. (An "accession number" is a number used to uniquely identify, or "access", items in a collection such as in a museum, library, or database.) This particular accession number is for a DNA sequences from a type of *Salmonella*. More on that shortly.

   After clicking on the "Search" button (to the right of where you entered the accession number) you will come to a page with a link that says Download. Clicking this will download a file called U81861.fa to your computer (note that the browser may require to you choose a filename, in which case you can enter U81861.fa and save to a location of your choosing).

   The file is in FASTA format and consists of a header line containing information about the sequence (the part beginning with a "¿" character) followed by multiple lines of the DNA sequence.

   You should move this file your homework directory for this homework assignment.

   Experimenting with `Python`'s file input syntax in the interpreter

   The file `goodStuff.txt` should be in the scripts directory along with `load.py`. This file just contains a bit of text that we'll play with before using the FASTA data that you downloaded a moment ago.

   This file can be viewed with a text editor such as the one you are using to view `Python` files. Open it now to take a look. The next task is to load this file into `Python`. You can read this file by cutting and pasting the

following text into the Python interpreter (note that you must open the interpreter from the directory that contains `goodStuff.txt`):

reading a file in Python

```
f = open("goodStuff.txt") # open the file
linesList = f.readlines() # read in the file as a list of its lines
f.close() # close the file
```

The above commands tell `Python` to open the file, read its lines into a list that we've named `linesList`, and then close the file. So far so good. But when we look at that list in `Python`, here's what we get:

contents of `linesList`

```
>>> linesList
["Turtles\n","Groodies\n","Spam\n"]
```

What are the "" characters about? These are are "newline characters" - they are extra characters that tell Python to start a new line after this string. The text file `goodStuff.txt` has three lines, and on the end of every line it has a newline character. When you view a file like `goodStuff.txt` with a text editor, you probably won't see the `""` chacracters. Instead, you'll see the actual new lines after each string. However when we load a text file like this into `Python`, it loads up everything including those annoying newline characters.

It is common in computing applications to manipulate our data slightly before we use it. In this case, we want to get rid of the newline characters. One easy way to remove them is to slice off the last character of each line using Python's slicing notation. Then, we'll store the slightly truncated line in a new list. Again you can cut and paste the following into the interpreter:

getting rid of newlines

```
newList=[]  # A new list!
for line in linesList:  # for each line in our linesList...
    newList.append(line[:-1])  # ... slice off the last character and
                               # append the truncated string to the newList
```

The above give you an example of how to use `Python`'s file input syntax to load things from a file. Now we'll use it to read a fasta file.

Loading a FASTA file: the `loadSeq(fileName)` function

FASTA files are actually just text files, and we can use the above syntax to load them.

Begin a new homework file called `load.py` (if you haven't already). We will now create a function called `loadSeq(fileName)` in this file. Note that `load.py` should not contain anything else besides this function (i.e. the practice stuff from the section above should not be included there.)

`loadSeq` will take one variable as input: a string with the name of the FASTA file that we'll be loading. This way we can use `loadSeq` to load any FASTA file we want, not just the one that we downloaded a moment ago. This function should return a single string (not a list!) which is all of the DNA in the lines of the input file.

Keep in mind that the input file is a FASTA format file. So, its first line contains information that is not part of the DNA. The subsequent lines are the actual DNA broken up over multiple lines. Your `loadSeq` function can start by using the same three lines below to read in the file and get the list of all of the lines in the file. Except, of course, now we're not necessarily reading `goodStuff.txt`, but rather the file whose name is specified in the string `fileName`.

getting rid of newlines

```
f = open(fileName) # open the file
linesList = f.readlines() # read in the file as a list of its lines
f.close() # close the file
```

But now what you want to return is a single string! So, you'll need a for loop and you'll need to slice. The idea is very similar to the for loop that we wrote above, but that one constructed a list and it included the first line. Now, we want to return a single string and we don't want the first line.

You can test your function on the following two FASTA files: `test1.fa` and `test2.fa`, which you will find in the scripts folder. test1.fa looks like this:

test1.fa

```
> Test fasta file 1
AGGTCTGTCAACCGTTTCAGTACA
ACCTAGCCTACCCTGCTAACTAGA
```

Here's our loadSeq function at work:

loadSeq

```
>>> loadSeq("test1.fa")
"AGGTCTGTCAACCGTTTCAGTACAACCTAGCCTACCCTGCTAACTAGA"
>>> loadSeq("test2.fa")
"CCGTCCATTCGACGGATA"
```

You have now created a module, which other python programs can use. `load.py` contains just one function. When it is imported by another program using import load this function will be brought into that program's memory space. (A python module can contain more than one function, but in this case we have only the one.)

Because `load.py` will be used as a module, it is important not to have other extraneous things included in it (e.g. no lines loading `goodStuff.txt` etc.)

Loading `U81861.fa`

You can now load the `U81861.fa` file you downloaded earlier. Be sure it is located in the same directory as your `load.py` file.

loading the fasta file

```
>>> salSeq = loadSeq("U81861.fa")
```

Next determine the length of this sequence. You can do this using the built-in len function. It takes a string or list as input and returns its length. For example:

using `len`

```
>>> len("spam")
4
```

Include the length of the U81861 sequence as a comment at the top of your `load.py` file.

4. **DNA manipulation**: In this problem, you'll write several helper functions for manipulating DNA sequences. These functions will be useful in subsequent problems where you'll analyze DNA data in various ways. All of your code for this problem should be in a file called `dna.py`.

Base complements

Write a function `compBase(N)` that takes a string that is a single DNA base `"A"`, `"G"`, `"T"`, or `"C"` as input and returns the base that is complementary to it. Here are some examples:

compBase(N) in action

```
>>> compBase("A")
'T'
>>> compBase("G")
'C'
```

Reverse complements

First, write a function called `reverse(s)` that takes in the string `s` and returns a string that is the reverse of `s`. Please use a for loop here, and do not use the built-in syntax `s[::-1]` (which is another way to reverse string). Here's an example of your reverse function in action:

reverse in action

```
>>> reverse("spam")
'maps'
```

In your for loop you can loop either over the characters of the input or over the indices. Both ways will work.

Next, write a function `reverseComplement(DNA)` that takes a DNA string as input (in 5' to 3' order) and returns the sequence of its complementary DNA strand, also in 5' to 3' order. This is called the reverse complement. This function will call both your reverse function and your `compBase` function for help. Here's an example:

reverse in action

```
>>> reverseComplement("TTGAC")
'GTCAA'
```

DNA to amino acids

The objective of this problem is to write a program that converts from DNA to a sequence of amino acids. Make sure you are in your scripts directory, where you will find a file called `aminoAcids.py`. Include the following line at the top of your dna.py file.

importing aminoAcids.py

```
from aminoAcids import *
```

This allows all the functions in your `dna.py` file to access the good stuff in the `aminoAcids.py` file. Just to see what's in there, run your `dna.py` file now. At the `Python` interpreter, then type:

example

```
>>> aa
['F', 'L', 'I', 'M', 'V', 'S', 'P', 'T', 'A', 'Y', '|', 'H', 'Q', 'N', 'K', 'D', 'E',
'C', 'W', 'R', 'G']
```

Notice that `aa` is the list of 20 amino acids. Next, type:

example

```
>>> codons
```

What you see now is a list of lists! That's weird. Notice that the first list in the codons list is `['TTT', 'TTC']`. This is telling us that that those two codons both code for the amino acid `'F'`. How do we know it's

'F'? Because 'F' is the first string in the aa list above. The next list in the codons list is ['TTA', 'TTG', 'CTT', 'CTC', 'CTA', 'CTG']. That's a big one! These six codons all code for the amino acid 'L', the second element in the aa list, etc. Your first task is write a function called amino(codon) that takes as input a codon string (a string of three letters) and returns the corresponding amino acid. One way to do this is to use a for loop that marches from 0 to the length of the aa list. For each index in that range, we can check if the codon is in the codon list at that index. Then, you're nearly done!

Keep in mind that all functions that you write in dna.py can access the lists aa and codons. Here's an example of a showOff function that uses the list aa.

showOff function

```
def showOff():
  ''' Prints all of the amino acids '''
  print "I know ALL the amino acids!"
  for amino in aa:   # <-- aa is available to us because we've imported aminoAcids
    print "Here's an amino acid ", amino
```

Next, write a function codingStrandToAA(DNA) that takes a sequence of DNA nucleotides from the coding strand and returns the corresponding amino acids as a string. Assume that the reading frame begins with the first base and continues on to the last nucleotide (i.e., codingStrandToAA doesn't need to worry about start and stop codons). This function should call your amino function for help!

You might find it useful to use a range of indices here again. In particular, note that while range(0, 10) will give you the list of all integers from 0 to 9, the variant range(0, 10, 2) will give you the range of numbers from 0 to 9 in increments of 2, e.g., [0, 2, 4, 6, 8]. Proceeding in increments of 3 will allow you to march from the start of one codon to the start of the next one!

You'll also find it useful to use the in thing! The word in is used in a few different ways in Python. It's used in for loops as in:

for loop using in

```
for food in ["spam", "hot dog", "hamburger", "arugula"]:
```

But it's also used to test if something is in a list or a string as in:

using in

```
>>> if "spam" in "I really love spam with chocolate": print True
True
>>> if 42 in [1, 6, 7, 42, 9]: print "Yup!"
"Yup!"
```

This latter use of in can be useful, for example, in determining whether or not a codon string is in a particular list of codons! Finally, here are some examples of what your codingStrandToAA function should be doing:

codingStrandToAA

```
>>> codingStrandToAA("AGTCCCGGGTTT")
'SPGF'
>>> codingStrandToAA("ATGCAACAGCTC")
'MQQL'
```

5. **Finding ORFs**: *All of your code for this problem should be in a file called* orf.py.

   Finding genes in DNA is a fundamental problem in biology. After all, only a few percent of human DNA actually contains protein-coding genes. Sifting through the more than 3 billion base pairs to find the parts that are most likely to be genes requires some computational craftiness. That's where we're headed! In this problem you'll write Python functions to find the longest open reading frame (ORF) in a DNA sequence.

This is closely related to the problem of finding all the genes in a sequence, which is the topic of the final problem for this part.

You'll make use of some of the functions that you wrote in the last problem. To get those functions, put the `dna.py` homework file in the same directory as the file for this problem, `orf.py`. Then, include the following line at the top of `orf.py`:

importing dna.py
```
from dna import *
```

This will allow you to access those functions in your current program.

Finding the first open reading frame

Recall that an open reading frame (ORF) is the stretch of sequence between a start codon (with the sequence "ATG") and the next in frame stop codon. By "in frame" we mean that it is in a position that is a multiple of 3 nucleotides away.

For example, take a look at the string: ATGCATAATTAGCT. There is an "ATG" at the beginning. Then there are two codons, "CAT" and "AAT", and then a stop codon "TAG". So "ATGCATAAT" is an open reading frame in this string. You might think for a moment that "ATGCATAA" is also an answer, but it's not an open reading frame because the "TAA" at the end of it is not "in frame" with the start codon "ATG" - that is, it's not a multiple of three nucleotides away from the leading "ATG".

Your first challenge is to write a function that finds the open reading frame given a sequence that begins with an "ATG". In this problem you need only operate on the forward sequence, and not consider its reverse complement.

Write a function called `restOfORF(DNA)` that takes as input a DNA sequence written 5' to 3'. It assumes that this DNA sequence begins with a start codon "ATG". It then finds the next in frame stop codon, and returns the ORF from the start to that stop codon. The sequence that is returned should include the start codon but not the stop codon. If there is no in frame stop codon, `restOfORF` should assume that the reading frame extends through the end of the sequence and simply return the entire sequence.

To this end, you will need to determine if a particular codon is a stop codon. Imagine that you have a string named codon and you wish to test if it is a stop codon, that is one of `'TAG', 'TAA'`, or `'TGA'`. You could do this:

example
```
if codon == 'TAG' or codon == 'TAA' or codon == 'TGA':
    blah, blah, blah
```

Or, better yet, you could use in this way:

example
```
if codon in ['TAG', 'TAA', 'TGA']:
    blah, blah, blah
```

Here are some examples of `restOfORF`:

restOfORF example

```
>>> restOfORF("ATGTGAA")
'ATG'
>>> restOfORF("ATGAGATAAG")
'ATGAGA'
>>> restOfORF("ATGAGATAGG")
'ATGAGA'
>>> restOfORF("ATGAAATT")
'ATGAAATT'
```

Note that in the last example there is no in frame stop codon, so we got back the whole string. Your `restOfORF` can be written with a for loop. It will be a nice short function.

Finding ORFs part 2

Next, you will write functions that can find open reading frames in sequences that don't begin with an ATG. These functions will search for ATGs and then call `restOfORF` to get the corresponding open reading frames.

Consider some sequence of interest. Imagine we start at the 0 position and count off in units of 3 nucleotides. This defines a particular reading frame for the sequence. Here is an illustration, where alternating +++ and — are used to indicate the units of 3 nucleotides.

codon demarcation

```
CAGCTCCAATGTTTTAACCCCCCCC
+++---+++---+++---+++---+
```

Considering just the given sequence (and not the reverse complement), we can define two other reading frames on this sequence, starting at either the 1 or the 2 positions.

codon demarcation

```
CAGCTACCATGTTTTAACCCCCCCC
-+++---+++---+++---+++---

CAGCTACCATGTTTTAACCCCCCCC
--+++---+++---+++---+++--
```

Every open reading frame between an ATG and a stop must fall in one of these three reading frames. A useful way to look for genes involves searching each of these frames separately for open reading frames.

`oneFrame`

Write a function `oneFrame(DNA)` that starts at the 0 position of DNA and searches forward in units of three looking for start codons. When it finds a start codon, `oneFrame` should take the slice of DNA beginning with that "ATG" and ask `restOfORF` for the open reading frame that begins there. It should store that sequence in a list and then continue searching for the next "ATG" start codon and repeat this process. Ultimately, this function returns the list of all ORFs that it found.

Here are some examples of `oneFrame` in action:

oneFrame example

```
>>> oneFrame("CCCATGTTTTGAAAAATGCCCGGGTAAA")
['ATGTTT', 'ATGCCCGGG']

>>> oneFrame("CCATGTAGAAATGCCC")
[]

>>> oneFrame("ATGCCCATGGGGAAATTTTGACCC")
['ATGCCCATGGGGAAATTT', 'ATGGGGAAATTT']
```

`longestORF`

Next, you will write a function `longestORF(DNA)` that takes a DNA string, with bases written 5' to 3', and returns the sequence of the longest open reading frame on it, in any of the three possible frames. This function will not consider the reverse complement of DNA. It shouldn't take much work to write `longestORF` given that you've already written `oneFrame`.

Consider the one sequence example from above:

example

```
>>> DNA="CAGCTCCAATGTTTTAACCCCCCCC"
```

We can look at the three frames of this sequence by slicing off 0, 1 or 2 base pairs at the start:

example

```
>>> oneFrame(DNA)
[]
>>> oneFrame(DNA[1:])
[]
>>> oneFrame(DNA[2:])
['ATGTTT']
```

Each call to `oneFrame` will produce a list. You can then combine the lists from the three calls, identify the largest ORF and return it. To combine two lists, you can do the following:

combining lists example

```
>>> aList=[1,4]
>>> bList=[5,6]
>>> combinedList=aList+bList
>>> combinedList
[1,4,5,6]
```

Also, remember that if you have a string and want to know its length, you can use the built-in `len` function. Here are some examples of `longestORF`:

longestORF example

```
>>> longestORF('ATGAAATAG')
'ATGAAA'
>>> longestORF('CATGAATAGGCCCA')
'ATGAATAGGCCCA'
>>> longestORF('CTGTAA')
''
>>> longestORF('ATGCCCTAACATGAAAATGACTTAGG')
'ATGAAAATGACT'
```

And finally: how long is the motB gene?

The *Salmonella* sequence that we downloaded in the loading sequences problem contains several genes. Finding all of them is a computational challenge that we'll address in the final problem for this part. For now, we'll just find the longest ORF on the forward strand. This is the motB gene that codes for part of the flagellar motor.

xample

```
>>> from load import *
>>> salSeq = loadSeq("U81861.fa")
>>> motBDNASeq = longestORF(salSeq)
>>> motBProtSeq = codingStrandToAA(motBDNASeq)
```

As a comment at the top of your `orf.py` file: indicate how long the motB gene is both in nucleotides and in amino acids. Include the sequence of the protein (the amino acid sequence).