Introduction

The third assignment for 600.112: Introduction to Programming for Scientists and Engineers finally delves into our first application area: basic molecular biology. Instead of cutting up frogs and pigs, we will mostly cut up and otherwise manipulate strings: sequences of characters. And as far as anyone knows, strings do not feel pain.

There are three things to do: First you’ll write a program that both validates DNA sequences and then computes their reverse complements. Second you’ll write a program that both validates DNA sequences and then simulates the process of “cutting” DNA using restriction enzymes. Third you’ll write a program that both validates mRNA sequences and then computes the primary structure of the protein a ribosome would construct for it.

There are detailed submission instructions on Piazza which you should follow to the letter! You can lose points if you create more work than necessary for the graders by not following the instructions.

Background

Molecular biology tries to understand the various chemical and biological processes that make cells work the way they do in living organisms. At the bottom of many of these processes we find two kinds complex molecules: nucleic acids and proteins. Proteins are (roughly) responsible for what an organism is and does, whereas nucleic acids are (roughly) responsible for encoding the information necessary to produce proteins.

There are many different kinds of proteins: Structural proteins are found in skin and hair; ligand-binding proteins facilitate, for example, oxygen transport; enzymes speed up certain chemical reactions, for example as part of the digestive system. Proteins are chains of simpler molecules called amino acids (or residues). Despite the wide variety of biological functions that proteins serve, most of them are built from only 20 different amino acids. Typical proteins contain about 300 of these amino acids.

There are two kinds of nucleic acids: deoxyribonucleic acid (DNA) and ribonucleic acid (RNA). DNA is a double-chain made up of two simple chains

*Disclaimer: This is not a course in physics or biology or epidemiology or even mathematics. Our exposition of the science behind the projects cuts corners whenever we can do so without lying outright. We are not trying to teach you anything but computer science!

1. A molecule is a group of atoms held together by chemical bonds. Although it won’t matter for this assignment, we’ll assume that you know what an atom is.
called *strands*. Each strand is in turn made up of simpler molecules called *bases* (or *nucleotides*). There are only four different bases: adenine (A), guanine (G), cytosine (C), and thymine (T). The two strands hold together because each base in one strand bonds to a base in the other strand: A bonds to T, C bonds to G. We say that A and T (C and G) **complement** each other or that A and T (C and G) are **complementary bases**. Each strand also has an **orientation** which by convention starts at the 5’ end and finishes at the 3’ end.\(^2\) RNA is similar to DNA in some respects, however RNA is typically single-stranded and thymine (T) is replaced with uracil (U). Also there are several different kinds of RNA that serve different purposes.

On a given DNA molecule, certain contiguous stretches called **genes** encode information for building proteins. Three bases of the DNA molecule correspond to one amino acid for the protein to be built, and each such “triplet” is called a *codon*. There are \(4^3 = 64\) possible “triplets” but only 20 amino acids. So while some amino acids have multiple codons, other “triplets” correspond to no amino acid at all: instead they indicate the end of a gene. A **promoter** region before the gene allows proteins that synthesize RNA to recognize where to start. The protein then makes a copy of the gene onto an RNA molecule called **messenger RNA** (mRNA).\(^3\) The mRNA molecules are built from their 5’ end to their 3’ end while the DNA strand is read from the 3’ end to the 5’ end. This process is called **transcription**. The actual proteins are produced by “biomolecular machines” called **ribosomes** according to the instructions carried by the mRNA. As you have probably guessed, ribosomes themselves are made out of proteins. Ribosomes also use two other forms of RNA, **ribosomal RNA** (rRNA) and **transfer RNA** (tRNA). This process is called **translation**.

The so-called “central dogma” of molecular biology explains why all of this is important:

\[
\begin{align*}
\text{DNA makes RNA,} \\
\text{RNA makes proteins,} \\
\text{proteins make us.}
\end{align*}
\]

**Figure 1 Output of the complement.py program.**

Input in file complement.dat:

\[
\begin{align*}
\text{ACTA} \\
\text{XENON} \\
\text{GATTACA} \\
\text{GGGGAGGAGGGTTTTAGGAAGTT}
\end{align*}
\]

Output:

\[
\begin{align*}
\text{TAGT} \\
\text{invalid input} \\
\text{TGTAATC} \\
\text{AACTTCCTAAAACCCTCCTCCCC}
\end{align*}
\]

So by understanding DNA, RNA, proteins, and the processes between them, we can understand—and in theory even improve—“us”.

1 **Reverse Complements (20%)**

The first program you will write first validates a DNA sequence and then computes and prints its reverse complement. Please call your program `complement.py` and nothing else! Figure 1 shows what the output of your program will look like given the input above it.

For this and the following programs, you will need to use *if instructions* and *functions* as discussed in lecture. This is in addition to your basic understanding of expressions, variables, for and while loops, etc. We will, however, lead you through the problems rather slowly and with a lot of advice on how to proceed, so you should be okay as long as you follow along diligently.

2. If you want to know why these things are called “the 5’ end” and “the 3’ end” you should take a course in molecular biology. All we care about here is that an orientation exists.

3. Strictly speaking this mechanism is only valid for some organisms. Again we defer the details to a real course in molecular biology.
Before you can do anything else, you’ll need a very basic first version of your program. We’ll have to read input data from the file `complement.dat` and just to see that we’re doing the right thing for now, we’ll print each line of that file back out. So here is what that first version could look like:

```python
def main():
    data = open("complement.dat")
    for line in data:
        print(line)
    data.close()
main()
```

If you run this version, you will notice that we have a blank line between each potential DNA sequence. The reason for this is that there is an “invisible” character at the end of each line we read that represents the end of that line in the file. When we read a file line-by-line using a for loop, that special character will be included at the very end of the string called `line` in our program. The print instruction add another “end-of-line” on its own, resulting in those empty lines you see in the output.

Since the “end-of-line” character is considered whitespace, we can use the `strip` operation on strings to get rid of it as follows:

```python
def main():
    data = open("complement.dat")
    for line in data:
        line = line.strip()
        print(line)
    data.close()
main()
```

We did something strange here: we defined the two functions `valid` and `complement` but we only put a `pass` instruction into their bodies. In Python, `pass` is an instruction that does nothing at all, but you can put it into places where you don’t know what to do yet; if we didn’t put `pass` into those functions, Python would complain that there is no code to go into the functions and we wouldn’t be able to run our program. This way, however, we can still run it to at least see that we didn’t make any programming errors: the program, while not doing anything new, at least still works as before.

What we can do now is “programming by wishful thinking” which is actually a better idea than what it first sounds like (at least if you do it correctly). Let’s simply assume that the functions already work as expected! So let’s assume that `valid` returns `True` if the string `dna` is a valid DNA sequence and `False` otherwise. Further let’s assume that `complement` returns the string that’s the reverse complement of the sequence we pass in as `dna`. Under those assumptions, what would the rest of the main program look like?

```python
def valid(dna):
    pass

def complement(dna):
    pass

def main():
    data = open("complement.dat")
    for line in data:
        line = line.strip()
        print(line)
    data.close()
main()
```

Now the output should be more what we expected initially, just the same as the input.

With this basic program working, we can think about what else we have to do for this problem. There are two remaining tasks: We need to check each line of input and see if it represents a valid DNA sequence. If the input is not a valid DNA sequence, we just print out invalid input for it and go on to the next input. If the input is a valid DNA sequence however, we need to compute and print its reverse complement. Just looking at these two things we have left to do, we can guess that we need at least two functions: one to decide whether an input is valid, and one to compute the reverse complement:

```python
def valid(dna):
    pass

def complement(dna):
    pass

def main():
    data = open("complement.dat")
    for line in data:
        line = line.strip()
        print(line)
    data.close()
main()
```
def complement(dna):
    pass

def main():
    data = open("complement.dat")
    for line in data:
        line = line.strip()
        if valid(line):
            print(complement(line))
        else:
            print("invalid input")
data.close()

main()

Note that this is simply an almost literal translation of what we said earlier: if the input is a valid DNA sequence, we print its reverse complement, otherwise we print invalid input.

If we run this version of the program, it will print invalid input for every single line of input. Since we didn’t actually write any code for valid and complement yet, this is actually better than we could expect. However it’s still a good idea to understand why this happens. Remember that I mentioned the value None before, which is the Python value that stands for “no value”? What happens is that when we call valid inside the if instruction, Python needs to replace that call with some value. But since our version of valid just contains a pass the function doesn’t produce a value. So Python “slips in” the value None for us, and it turns out that in a boolean expression like the one expected as the condition of an if or while, None means the same thing as False.

Let’s fix this by actually writing the proper valid function. The function gets a string and it has to decide whether the string represents a DNA sequence or not. A string that is a valid DNA sequence can only consist of the characters ACTG and nothing else. So what we can do is go through the string dna one character at a time; if we find any character that’s not one of ACTG we can return False even before we are finished with the string; if, one the other hand, we get to the end of the string and never saw anything but ACTG, the we can return True. Here is the code:

```python
def valid(dna):
    for c in dna:
        if not c in "ATGC":
            return False
    return True
def complement(dna):
    pass
def main():
    data = open("complement.dat")
    for line in data:
        line = line.strip()
        if valid(line):
            print(complement(line))
        else:
            print("invalid input")
data.close()

main()
```

If we run this version of the code, it’ll print invalid input for the second input and None for the remaining three inputs. This again makes sense: When we call complement for a valid DNA sequence, the function doesn’t yet produce a value so Python once again “slips in” the value None for us. But at least we’re now doing the correct thing for the invalid input!

This leaves the complement function. To complement a DNA sequence we have to replace each base by its complementary base, so we replace A by T and vice versa, as well as C by G and vice versa. The reverse complement of a DNA sequence is the complement in reverse, so we also have to turn the resulting string around: what used to be the end is now the beginning and vice versa.

This is complicated enough to once again write a new function. Let’s write a function partner that given a base will return the complementary base; if what we’re given is not a valid base, we’ll just return the base X for “invalid base”. Here’s the code (we’ll see how to write this much nicer next week):

```python
def partner(c):
    if c == 'A':
        return 'T'
    elif c == 'T':
        return 'A'
    elif c == 'C':
        return 'G'
    elif c == 'G':
        return 'C'
    else:
        return 'X'
```
Once we have the `partner` function, the `complement` function becomes rather straightforward: We start with an empty string for our result; we then go over the `reverse` string character by character; for each character in `dna` we append the complementary `partner` to the result string; and once we’re done, we return that result. If you run the program now, it should produce the expected output. Here is the code:

```python
... 
def complement(dna):
    result = ""
    for c in reversed(dna):
        result = result + partner(c)
    return result
... 
```

If the enzyme “finds” such a pattern, it “cuts” after the G and before the first A in the pattern. For a DNA sequence such as `GGGAAATTCGG` this means that one cut exists, and that cut separates the DNA into `GGG` and `AATTCGG`.

The basic structure of the program is similar to Problem 1, indeed you can even copy the `valid` function from there. The main difference is what we do once we have a valid DNA strand to process: before we complemented the strand, now we have to find and print all cuts that EcoRI would make. Here’s some code to start with:

```python
... 
def valid(dna):
    for c in dna:
        if not c in "ATGC":
            return False
    return True
... 
```

4. We should actually consider both strands of the DNA to do this “properly” but we’ll just ignore that complication here.

---

2 Cutting DNA (40%)

The second program you will write first validates a DNA sequence and then simulates the process of “cutting” DNA using restriction enzymes. Please call your program `cuts.py` and nothing else! Figure 2 shows what the output of your program will look like given the input above it.

There are various restriction enzymes that end up cutting DNA at different places. The enzyme we’ll simulate here is called EcoRI (“eco-R-one”) and cuts DNA at recognition sites with the pattern `GAATTC`.

**Figure 2 Output of the `cuts.py` program.**

Input in file `cuts.dat`:

ATTGAATTCATTGAATTCGGGG
XENON
ATTATTATTATTATGGGGG
GAA
GGGGGAAATTCCGGGGGGGGG

Output:

ATTGAATTCATTGAATTCGGGG
    |4
    |13
invalid input
ATTATTATTATTATGGGGG
GAA
GGGGGAAATTCCGGGGGGGGG
    |6

---

600.112: Introduction to Programming for Scientists and Engineers  Fall 2014
def main():
    data = open("cuts.dat")
    for line in data:
        line = line.strip()
        if valid(line):
            print_all_cuts(line)
        else:
            print("invalid_input")
data.close()

main()

It's probably a bad idea to start with the `print_all_cuts` function without first thinking about the problem a little more. First, how do you find the position of the recognition site itself? In lecture we saw the `find` function on strings that comes in handy here. Carefully look at the following Python Shell interaction:

```python
>>> "GGATCC".find("GAATTC")
-1
>>> "GGAATTCC".find("GAATTC")
1
>>> "GGAATTCGGAATTC".find("GAATTC")
1
>>> "GGAATTCGGAATTC".find("GAATTC", 2)
9
>>> "GGAATTCGGAATTC".find("GAATTC", 10)
-1
```

If the pattern we're looking for is not in the string, we get 
-1 from find. If the pattern is in the string, we get the position of the first matching character so 1 in this case. But what if there are multiple places in the string where the pattern matches? To account for this case, `find` takes an optional second parameter to indicate at what position in the string to start looking for a pattern. So if we find the first matching pattern at position 1, we can ask `find` to search again starting from position 2 to see if there is another match. In this case we find another match at position 9, so we ask again starting from position 10; now we get back 
-1 so we know that there are no further matches for the pattern in the string.

This process is rather similar to finding all the "cuts" in a given strand of DNA. I recommend that you write and test a function `cut(dna, start)` that gets a string representing a DNA sequence as `dna` and a position in that string from where to find the next cut. The function will return the position of the next cut starting at the start position; if there is no further cut, it will return 
-1 just like `find` does. Remember that the position of the cut is one greater than the position of the recognition site in the dna string!

Once you have the `cut` function, you can try your hand at the more complicated `print_all_cuts` function. The function starts simple enough by just printing the DNA sequence itself because (check the output in Figure 2 above!) the actual cuts will be printed on separate lines after the DNA sequence. You have to find a first cut (if there is one) by calling `cut`. Then, if you did indeed find a cut, you have to print the corresponding marker at the right indentation to line up with the first A in the recognition site pattern. Remember that you can multiply a string by a number, so for example the expression "A"*4 results in the string "AAAA"; this can be used for indentation. Also remember that you can convert an integer into a string suitable for appending to another string using the explicit `str` type conversion function. This should allow you to produce the correct output marking each cut. Note that if you don’t find a cut, you print nothing else after the DNA sequence itself. The “tricky” thing about the loop you have to write is that you need to keep track of the last cut position as well as the next cut position. You may have to play with this for a while and maybe talk the correct approach through with your TA.

3 Playing Ribosome (40%)

The third program you will write first validates an mRNA sequence and then computes the primary structure of the protein a ribosome would construct for it. Please call your program `ribosome.py` and nothing else! Figure 3 shows what the output of your program will look like given the input above it.
Figure 3 Output of the ribosome.py program.

Input in file ribosome.dat:

AUUGAAUUCAUUGAAUUCGGG
XENON
GAA
GGGGGAAUUCGGGGGGG
GAAUUUGAUUUUU

Output:

IleGluPheIleGluPheGly
invalid input
Glu
GlyGlyAsnSerGlyGly
GluPhe

We’ll provide you with a module called protein that contains a single function also called protein that can translate each codon of the mRNA into the correct amino acid or the special string STOP that indicates the end of a gene. Here’s an example session in the Python Shell:

```python
>>> import protein
>>> protein.protein("AGU")
'Ser'
>>> protein.protein("UGA")
'STOP'
```

Please use this function in your program by importing the module as we’ve shown above (don’t copy/paste the function into your code). There’s not much advice for this problem. Make sure you rewrite valid to check for RNA and not DNA bases, and make sure that your code runs down the codons only until it encounters a STOP codon or until the mRNA string itself doesn’t have enough characters for another codon left. The rest, as they say, is up to you.