Lesson 8

August 7, 2020

1 Lesson 8: Build Your Own Protein II

As a refresher, we learned the following in lesson 7: - Genes are made up of DNA and provide the information for our cells to carry out important biological processes that ultimately lead to phenotype - The DNA provides instructions for building proteins, which are made up of essential building blocks called amino acids. - Almost all proteins are made up of combinations and permutations of the same 20 amino acids. - DNA is composed of 4 different molecules called nucleotides or bases: adenosine, thymine, cytosine, and guanine. We abbreviate these as A, T, C, and G, respectively. - This triplet genetic code is actually conferred in the biology of using DNA to make proteins through an intermediate, RNA; RNA, or ribonucleic acid, is composed of three of the same nucleotides as DNA: adenosine, cytosine, and guanine, and with uracil replacing thymine. DNA is copied into RNA in a process called transcription. - RNA is then used as a template to make a protein in a process called translation. The transcription of DNA into RNA, which can then be translated into proteins is called The Central Dogma of biology.

For this next section, we'll make significant use of loops and lists. So let's recall some of the basics. Lists are objects that can store multiple items as shown below in the list: lesson8_chapters (Note: Lists can store text and integers / floats as well as more complex objects).

[1]:
```
lesson8_chapters = ['genetics refresher', 'lists & loops refresher', 'list␣
↪methods', 'dictionaries', 'checkpoint', 'build your own protein', 'summary']
```

The lists that we are working with are reasonable sizes, but in practical application, lists can be several magnitudes larger, so we need quick ways of referencing locations in a list and determining the overall size. This is demonstrated below with two functions, the first one prints the fifth item in the lesson8_chapters list (remember that lists are zero-indexed) and the second prints the length of the list.

[3]:
```
print(lesson8_chapters[0])
print(len(lesson8_chapters))
```

```
genetics refresher
```

When we have large lists, it is often helpful to iterate through the list automatically. This is where For loops cmoe in. Using the below syntax, we create a new variable called lesson_name which will represent one of the items in the lesson8_chapters list. Then it will print out each name.

Recall that is also possible to combine for loops with if statements. Scientists will often use if statements in for loops to search for a particular item within a large list.

[4]:
```
for lesson_name in lesson8_chapters:
     print('Lesson 8 contains the following', lesson_name)
```

```
('Lesson 8 contains the following', 'genetics refresher')
('Lesson 8 contains the following', 'lists & loops refresher')
('Lesson 8 contains the following', 'list methods')
('Lesson 8 contains the following', 'dictionaries')
('Lesson 8 contains the following', 'checkpoint')
('Lesson 8 contains the following', 'build your own protein')
('Lesson 8 contains the following', 'summary')
```

Using the terminal below, create a list called lesson8_chapters and print how many items are in the list that are not "checkpoint" or "summary".

In this case, pretend that you do not know how many items are in the list.

[7]:
```
lesson8_chapters = ['genetics refresher', 'lists & loops refresher', 'list␣
 ↪methods', 'dictionaries', 'checkpoint', 'build your own protein', 'summary']
lesson_count = len(lesson8_chapters)

for lesson_name in lesson8_chapters:
        if lesson_name == 'checkpoint' or lesson_name == 'summary':
            lesson_count = lesson_count - 1

print(lesson_count)
```

```
5
```

1.1 List Methods

Because lists are so useful in programming, Python has some useful built-in list functions that we can use to make our code more efficient. For example, to count the number of lysines in our protein we can simply write:

[9]:
```
my_protein = ['methionine','valine','leucine','serine', 'proline', 'alanine',␣
 ↪'lysine', 'threonine','asparagine', 'valine','lysine','alanine', 'alanine',␣
 ↪'tryptophan', 'glycine','lysine', 'valine', 'glycine','alanine']
print(len(my_protein))
print('There are ', my_protein.count('lysine'), 'lysines in my protein')
```

```
19
('There are ', 3, 'lysines in my protein')
```

Note the difference between the len() function and the method .count().

Another common list function is append. This adds on a new element to the end of an existing list. For example, let's say that we discover the next amino acid in the hemoglobin protein is a histidine, and we want to add this amino acid to our protein list. This is accomplished with a simple one-liner:

[10]:
```
my_protein = ['methionine','valine','leucine','serine', 'proline', 'alanine',␣
↪'lysine', 'threonine','asparagine', 'valine','lysine','alanine', 'alanine',␣
↪'tryptophan', 'glycine','lysine', 'valine', 'glycine','alanine']
my_protein.append('histidine')
print(my_protein)
```

```
['methionine', 'valine', 'leucine', 'serine', 'proline', 'alanine', 'lysine',
'threonine', 'asparagine', 'valine', 'lysine', 'alanine', 'alanine',
'tryptophan', 'glycine', 'lysine', 'valine', 'glycine', 'alanine', 'histidine']
```

If we want to see the last element in our list, to make sure that "append" worked, we can use indexing. Of course, we could index using the length of the protein (minus 1 - don't forget the 0-indexing!), to get the index of the last element:

There is an easier way, though! Python allows us to use negative numbers to index in from the back of the list. For example, to print the last element of our list, we can simply index in with -1!

Both methods are shown below:

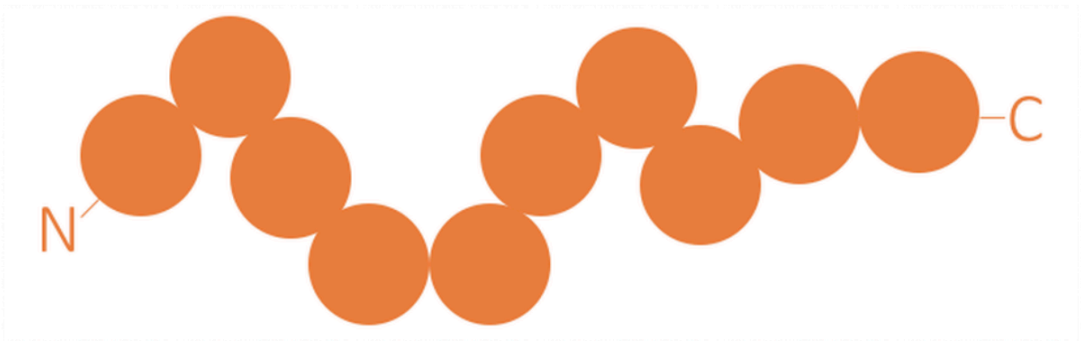
[11]:
```
my_protein = ['methionine','valine','leucine','serine', 'proline', 'alanine',␣
↪'lysine', 'threonine','asparagine', 'valine','lysine','alanine', 'alanine',␣
↪'tryptophan', 'glycine','lysine', 'valine', 'glycine','alanine']

print(my_protein[len(my_protein)-1])

print(my_protein[-1])
```

```
alanine
alanine
```

In biology, proteins are not symmetrical. That is, there is a defined start and end to each chain of amino acids at the structural level. Amino acids are linked in a chain that goes like this:



Notice that there is a N (which stands for a nitrogen atom) at the start of the amino acid chain and a C (which stands for carboxy) at the end of the chain. For this reason, we often refer to the start of a protein as the N-terminus and the end of a protein as the C-terminus. The negative indexing can be really useful to look at the C-terminus of a protein, where a lot of interesting biology can happen

1.2 Indexing to multiple elements

So far, we've only looked at the entire contents of the list or a single element. But what if we wanted to look at several consecutive elements of a list? It's easier than you think! To print the first three elements in a list, we can just index in with 0:3. With this syntax, the first number is the starting index of interest and the second number is the last index of interest, plus 1 (the first index's element will be included in the list, but the second element's element will not be).

[12]:
```
my_protein = ['methionine','valine','leucine','serine', 'proline', 'alanine',␣
↪'lysine', 'threonine','asparagine', 'valine','lysine','alanine', 'alanine',␣
↪'tryptophan', 'glycine','lysine', 'valine', 'glycine','alanine']

# Slicing - what if we wanted to see the first three amino acids?
print(my_protein[0:3])
```

```
['methionine', 'valine', 'leucine']
```

Print out the last 6 amino acids of our protein

[13]:
```
my_protein = ['methionine','valine','leucine','serine', 'proline', 'alanine',␣
↪'lysine', 'threonine','asparagine', 'valine','lysine','alanine', 'alanine',␣
↪'tryptophan', 'glycine','lysine', 'valine', 'glycine','alanine']
print(my_protein[len(my_protein)-7:len(my_protein)-1])
```

```
['alanine', 'tryptophan', 'glycine', 'lysine', 'valine', 'glycine']
```

Notice that we could have done this in a number of alternative ways. We could have looked at the list and printed each one individually, but this version ensures that the program will run regardless of the length of the list or its contents

1.3 Dictionaries

Let's talk about a new type of data structure. Think about a dictionary. Yes, a real, paper, heavy, old, musty dictionary (try to imagine the days before Google!). In the olden days, we had to look up the definition of a word we didn't understand in a dictionary. Every word in the dictionary is unique and associated with a definition. And it was actually pretty easy to use a dictionary; you could find the definition that you were looking for really fast (maybe not quite as fast as Google)! Just like paper dictionaries, Python has a data structure that works just like a dictionary - you can use a key (like a word you don't understand) to get information associated with that key (a value, like a definition of the word) really quickly. And it's even called a dictionary in Python!

A couple of important things to know about Python dictionaries:

- Dictionaries are unordered key-value pairs. This is where Python dictionaries differ from paper dictionaries; while a paper dictionary is in alphabetical order, Python dictionaries are not ordered at all. This means we can't use indexing, like in lists, to access the values of a dictionary (we have to use the keys!). But that doesn't mean that it's not fast to look up the value associated with a key!
- Dictionary keys all must be unique (think about all of the confusion that homonyms already cause for paper dictionaries!). But the values that they are associated with don't have to be unique (just like synonyms!).

- Dictionaries are especially useful when trying to associate one string with another (although keys and values can be other data types as well, including strings or integers).
- Dictionaries are not the same as defining a bunch of new variables! A variable name cannot be a string and a string cannot be a variable name. However, a dictionary key can be a string. Moreover, keys only access their paired value when called in the dictionary.

Remember the Genetic Code? A dictionary is the perfect data structure to store the Genetic Code, because we have unique keys (triplets of nucleotides, like ATG) that code for not-necessarily-unique values (amino acids, like methionine). We can create the start of this Genetic Code dictionary with this syntax:

[14]:
```
genetic_code_dict = { 'ATG': 'methionine', 'TTT': 'phenylalanine', 'CGG':␣
↪'arginine' }

print(genetic_code_dict['TTT'])
```

```
phenylalanine
```

Notice the new syntax to create a dictionary. We can store the entire dictionary in a variable, here called genetic_code_dict. We set this variable equal to a dictionary, denoted by the curly brackets {}. Then, we can define each key : value pair, using a colon : between them and a comma , to separate the pairs.

Sometimes the dictionaries we create can get pretty long, so it would be easy to forget what keys and values are included in the dictionary. Luckily, it is easy to print out the keys and values of a dictionary as a list:

[15]:
```
genetic_code_dict = { 'ATG': 'methionine', 'TTT': 'phenylalanine', 'CGG':␣
↪'arginine' }

print(genetic_code_dict.keys()) # Prints out keys of dictionary

print(genetic_code_dict.values()) # Prints out values of dictionary

print(len(genetic_code_dict))
```

```
['ATG', 'CGG', 'TTT']
['methionine', 'arginine', 'phenylalanine']
3
```

Now that we've created a dictionary with key : value pairs, we need to know how to "read" the dictionary by looking up values by their keys. This is accomplished easily with the syntax dictionary[key]!

1.4 Checkpoint

Find the amino acid associated with the codon TTT

[16]:
```

```

```
genetic_code_dict = { 'ATG': 'methionine', 'TTT': 'phenylalanine', 'CGG':␣
↪'arginine' }
print(genetic_code_dict['TTT'])
```

```
phenylalanine
```

That's right, you can access any part of a dictionary as long as you have the key! Lists and dictionaries share many of the same properties. Can you guess how to calculate the length of the dictionary?

1.5 Build Your Own Protein

[17]:
```
# Our genetic code dictionary
genetic_code_dict = { 'ATG': 'methionine', 'TTT': 'phenylalanine', 'CGG':␣
↪'arginine'}

# DNA sequence, already grouped into codons
dna_sequence = ['ATG','CGG','TTT','CGG','TTT']
amino_acids = []


# An empty list to hold the sequence of amino acids specified by the DNA␣
↪sequence
# Loop through each DNA codon
for codon in dna_sequence:

    # Look up the corresponding amino acid in a genetic code dictionary
    amino_acid = genetic_code_dict[ codon ]

    # Append the correct amino acid to our growing amino acid list (protein)
    amino_acids.append( amino_acid )

print(amino_acids)
```

```
['methionine', 'arginine', 'phenylalanine', 'arginine', 'phenylalanine']
```

For the last example, we started with a DNA sequence already grouped into codon triplets. This is pretty unrealistic, though. Generally, a DNA sequence that you would work with when coding is just a string of As, Cs, Ts, and Gs. The good news is that we can easily convert a string of DNA nucleotides into a list of codons. To do this, we need one more tool in our toolbox: modulo.

Modulo, although it sounds complicated, is just another word for the remainder left after division. If two numbers divide into one another with no fraction remaining, then those numbers are said to have "modulo 0." For example, 4 and 2 have modulo 0 because 4 is divisible by 2 with no remainder. 5 and 2, however, have modulo 1, because 5 divided by 2 is 2 with a remainder of 1.

We can calculate modulo in Python with the % symbol:

[18]:
```
print(4 % 2) # 4 modulo 2 is 0

print(5 % 2) # 5 modulo 2 is 1

print(0 % 2) # 0 modulo any number is 0!
```

```
0
1
0
```

So why are we on this modulo tangent? Because we want to divide a string of DNA nucleotides into a list of triplets! So we will use modulo 3 to check if we've reached the beginning of a new codon when reading through the DNA string. Remember that a string can be treated just like a list of characters; we can index into the string and divide up the string. So, we can convert a string of nucleotides into a list of codons:

[19]:
```
dna_sequence = 'ATGCGGTTTCGGTTT'

codons = [] # Create an empty list to eventually hold the codons

i = 0 # Start a counter at 0

# Move through each index in the DNA sequence string length
for i in range(0, len(dna_sequence)):
    if i % 3 == 0: # If the index is divisible by 3, then...
        # Index into the DNA sequence using i thru (i + 3) to get 3 nucleotides
        codons.append(dna_sequence[i:i+3])
print(codons)
```

```
['ATG', 'CGG', 'TTT', 'CGG', 'TTT']
```

What would happen if we didn't use the modulo function? We would triple-count all of the nucleotides and end up with the wrong protein!

[20]:
```
dna_sequence = 'ATGCGGTTTCGGTTT'

codons = [] # Create an empty list to eventually hold the codons
i = 0 # Start a counter at 0
# Move through each index in the DNA sequence string length
for i in range(0, len(dna_sequence)):
     # Index into the DNA sequence using i through (i + 3) to get 3 nucleotides
    codons.append(dna_sequence[i:i+3])

print(codons)
```

```
['ATG', 'TGC', 'GCG', 'CGG', 'GGT', 'GTT', 'TTT', 'TTC', 'TCG', 'CGG', 'GGT',
'GTT', 'TTT', 'TT', 'T']
```

Clearly, this isn't what we were trying to do - there are obviously too many codons in our solution to have come from the original DNA sequence. Thank goodness for the modulo function!

1.6 Summary

Combine our previous two calculations (creating a protein from a list of codons and creating a list of codons from a DNA nucleotide string) to generate a protein from a DNA nucleotide string. If you really want to challenge yourself, expand our genetic code dictionary to encompass more of the genetic code - and then build entirely your own protein! (Hint: If you're having trouble, add print statements to see what values variables are taking on!)

[ ]:
```
genetic_code_dict = { 'ATG': 'methionine', 'TTT': 'phenylalanine', 'CGG':␣
↪'arginine'}
dna_sequence = 'ATGCGGTTTCGGTTT'
```

You should now understand the following:

How to define a list data frame

Recognize appropriate situations to employ for loops

Implement for loops by index and element of a list

Design a program to create a protein from a nucleotide sequence