INST126

Jesse Anderson

Final Project

A collection of functions that aim to combine well-known bioinformatics theory, like the Needleman-Wunsch algorithm, with rudimentary but streamlined processes for genetic analysis

Functions explained, in order

amino_list
amino_string
codon_dictionary
global_alignment_matrix
transcription

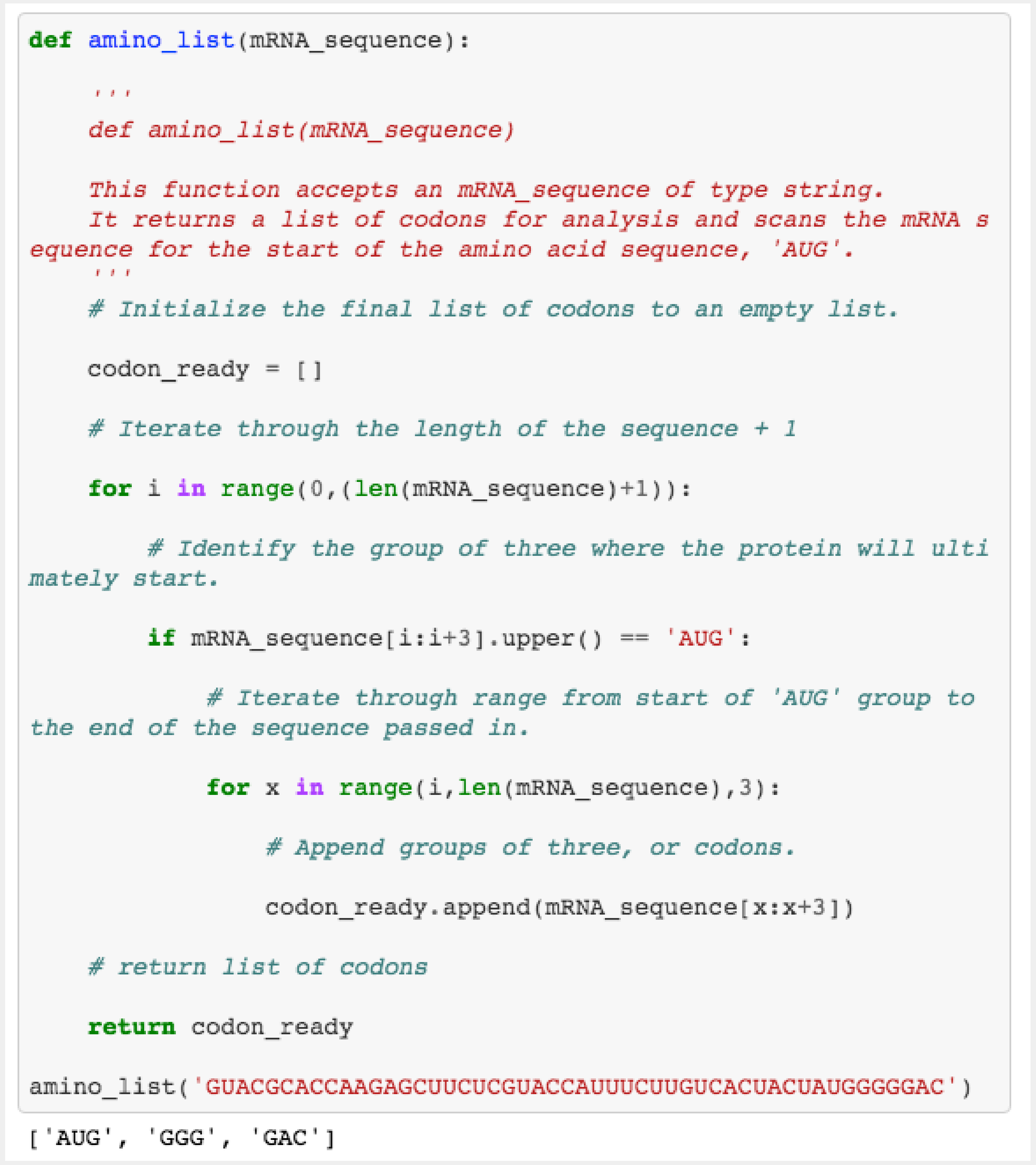
def amino_list(mRNA_sequence)

Parameters: a sequence of nucleotides of type string.

The goal: The goal of this function is to process a nucleotide sequence and convert it into a sequence of codons.

The logic (in brief): I iterated through all nucleotides in the sequence until I found a consecutive group of three that was equivalent to 'AUG' (met), which is the start of every amino acid sequence. This, in the code, is evidenced by [i:i+3] == 'AUG'. I used the *slicing*. Once I found this, I continued to interate through the sub-sequence after that i, which symbolized the index at which 'AUG' began. But, I made sure to use a step-size of 3 so that I could divide the sub-sequence further into groups of three. I appended these codons to my list that I return, called *codon_ready*.

The Code



```
def amino_list(mRNA_sequence):
    '''
    def amino_list(mRNA_sequence)

    This function accepts an mRNA_sequence of type string.
    It returns a list of codons for analysis and scans the mRNA s
equence for the start of the amino acid sequence, 'AUG'.
    '''
    # Initialize the final list of codons to an empty list.

    codon_ready = []

    # Iterate through the length of the sequence + 1

    for i in range(0,(len(mRNA_sequence)+1)):

        # Identify the group of three where the protein will ulti
mately start.

        if mRNA_sequence[i:i+3].upper() == 'AUG':

            # Iterate through range from start of 'AUG' group to
the end of the sequence passed in.

            for x in range(i,len(mRNA_sequence),3):

                # Append groups of three, or codons.

                codon_ready.append(mRNA_sequence[x:x+3])

    # return list of codons

    return codon_ready

amino_list('GUACGCACCAAGAGCUUCUCGUACCAUUUCUUGUCACUACUAUGGGGGAC')
```

```
['AUG', 'GGG', 'GAC']
```

def codon_dictionary(mRNA_sequence)

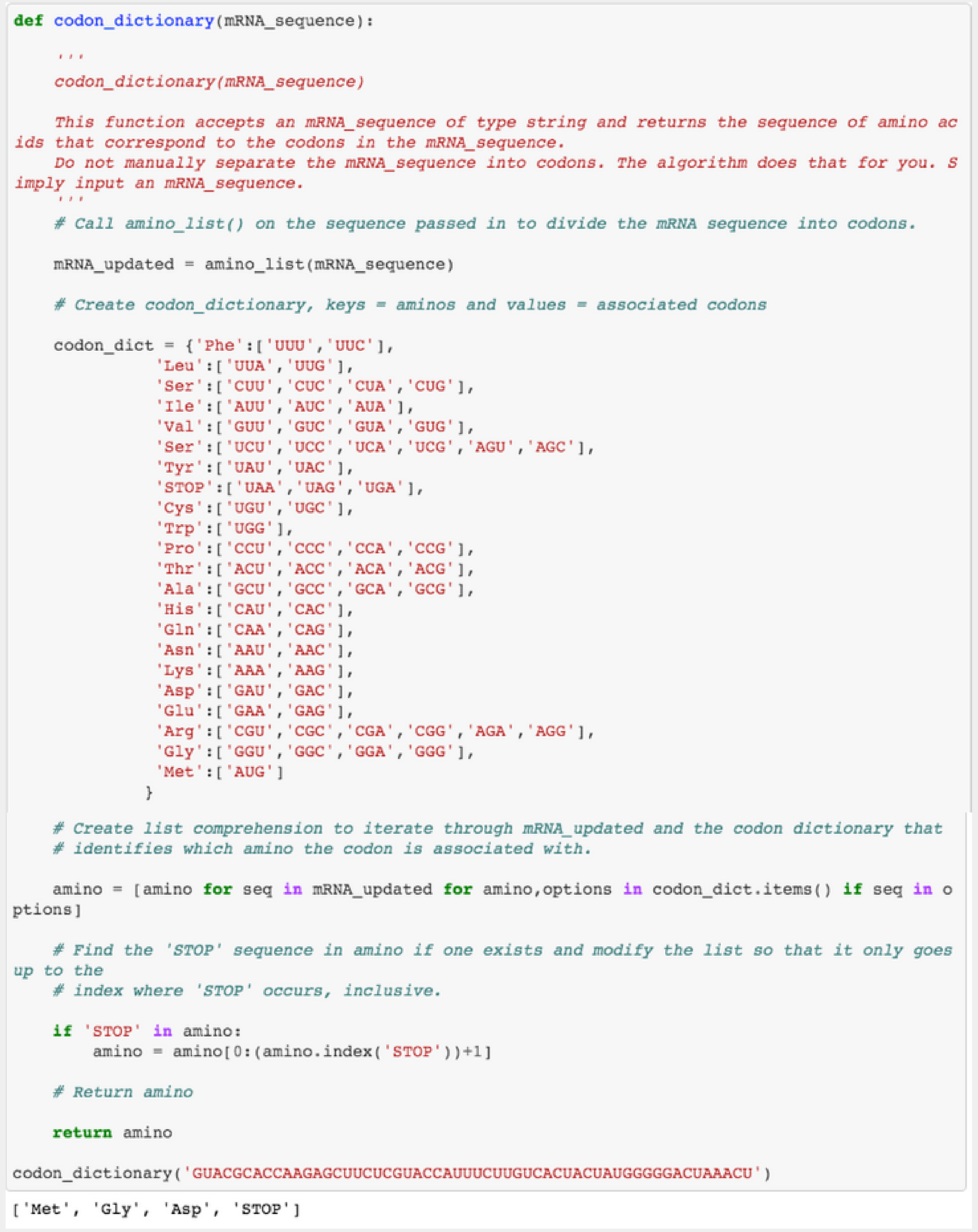
***Parameters*:** a sequence of nucleotides of type string.

***The goal*:** The goal of this function is to create a codon dictionary and convert a sequence of codons into a sequence of amino acids.

***The logic (in brief)*:** First I called the function amino_list on the mRNA_sequence submitted and stored the output in the variable mRNA_updated. Then I created a codon dictionary, which is a dictionary that has the amino acid name as every key and the associated codons in a list as the values.

I first iterated through each codon in mRNA_updated. Next, I created a nested loop and performed tuple unpacking on the codon dictionary that I created in the previous step. I then checked to see if the codon was in the list of codons associated with that amino acid. If it was, then I added that amino acid (or the key associated with the values) to the list.

The Code



```
def codon_dictionary(mRNA_sequence):

    '''
    codon_dictionary(mRNA_sequence)

    This function accepts an mRNA_sequence of type string and returns the sequence of amino ac
ids that correspond to the codons in the mRNA_sequence.
    Do not manually separate the mRNA_sequence into codons. The algorithm does that for you. S
imply input an mRNA_sequence.
    '''
    # Call amino_list() on the sequence passed in to divide the mRNA sequence into codons.

    mRNA_updated = amino_list(mRNA_sequence)

    # Create codon_dictionary, keys = aminos and values = associated codons

    codon_dict = {'Phe':['UUU','UUC'],
              'Leu':['UUA','UUG'],
              'Ser':['CUU','CUC','CUA','CUG'],
              'Ile':['AUU','AUC','AUA'],
              'Val':['GUU','GUC','GUA','GUG'],
              'Ser':['UCU','UCC','UCA','UCG','AGU','AGC'],
              'Tyr':['UAU','UAC'],
              'STOP':['UAA','UAG','UGA'],
              'Cys':['UGU','UGC'],
              'Trp':['UGG'],
              'Pro':['CCU','CCC','CCA','CCG'],
              'Thr':['ACU','ACC','ACA','ACG'],
              'Ala':['GCU','GCC','GCA','GCG'],
              'His':['CAU','CAC'],
              'Gln':['CAA','CAG'],
              'Asn':['AAU','AAC'],
              'Lys':['AAA','AAG'],
              'Asp':['GAU','GAC'],
              'Glu':['GAA','GAG'],
              'Arg':['CGU','CGC','CGA','CGG','AGA','AGG'],
              'Gly':['GGU','GGC','GGA','GGG'],
              'Met':['AUG']
             }

    # Create list comprehension to iterate through mRNA_updated and the codon dictionary that
    # identifies which amino the codon is associated with.

    amino = [amino for seq in mRNA_updated for amino,options in codon_dict.items() if seq in o
ptions]

    # Find the 'STOP' sequence in amino if one exists and modify the list so that it only goes
up to the
    # index where 'STOP' occurs, inclusive.

    if 'STOP' in amino:
        amino = amino[0:(amino.index('STOP'))+1]

    # Return amino

    return amino

codon_dictionary('GUACGCACCAAGAGCUUCUCGUACCAUUUCUUGUCACUACUAUGGGGGACUAAACU')
```

```
['Met', 'Gly', 'Asp', 'STOP']
```

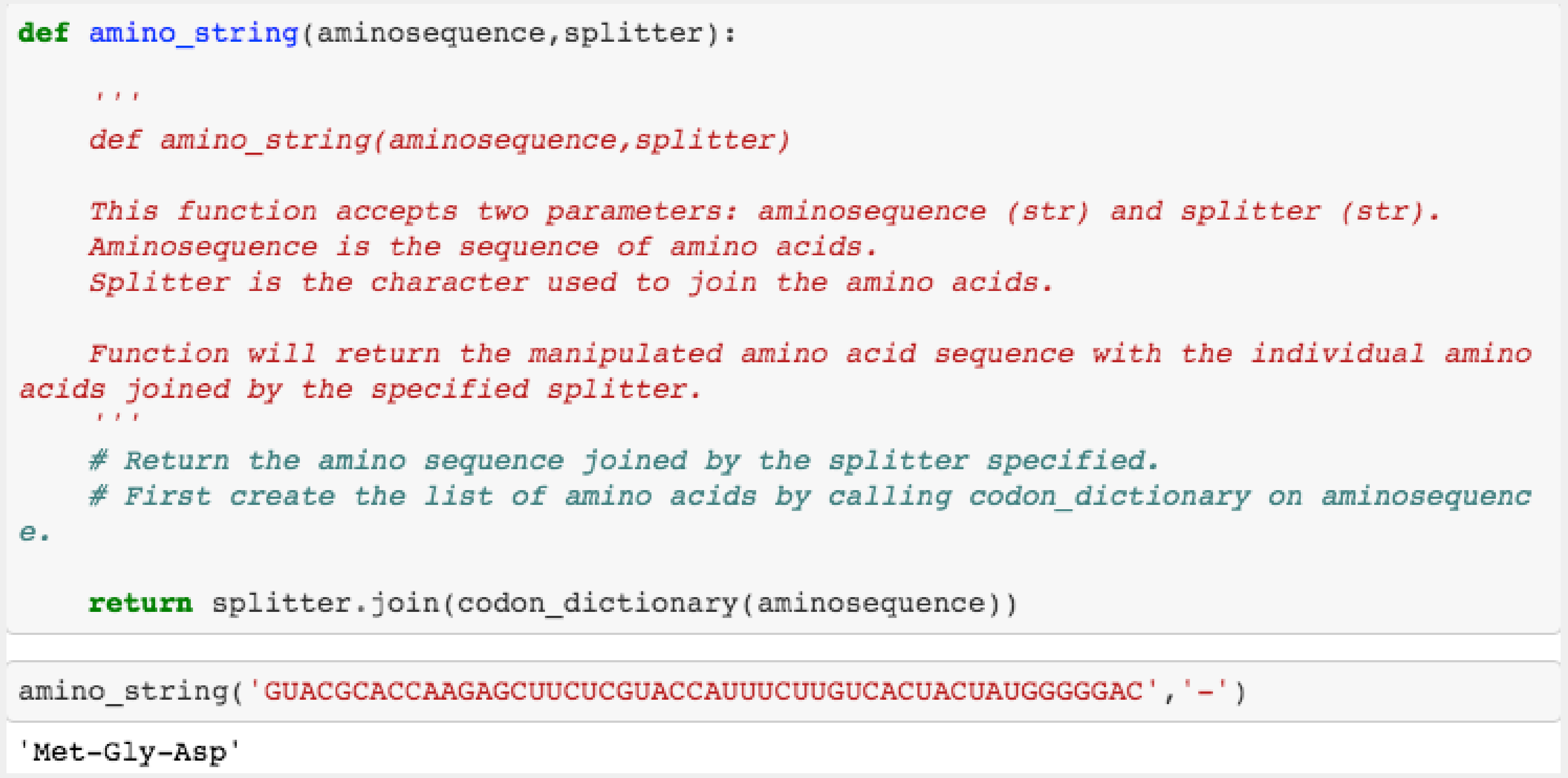
def amino_string(aminosequence,splitter):

Parameters: a sequence of nucleotides of type string and a splitter of type string.

The goal: The goal of this function is to allow the user to specify a divider between the amino sequences if they choose to do so. This would return something like: AMINO-AMINO-AMINO-AMINO-AMINO, for example, if splitter='-'.

The logic (in brief): I first called codon_dictionary( ) on the amino sequence passed in. This created a list of amino acids, which I then joined with the splitter specified using the .join( ) built-in function.

The Code



```
def amino_string(aminosequence,splitter):
    '''
    def amino_string(aminosequence,splitter)

    This function accepts two parameters: aminosequence (str) and splitter (str).
    Aminosequence is the sequence of amino acids.
    Splitter is the character used to join the amino acids.

    Function will return the manipulated amino acid sequence with the individual amino
acids joined by the specified splitter.
    '''
    # Return the amino sequence joined by the splitter specified.
    # First create the list of amino acids by calling codon_dictionary on aminosequenc
e.

    return splitter.join(codon_dictionary(aminosequence))
```

```
amino_string('GUACGCACCAAGAGCUUCUCGUACCAUUUCUUGUCACUACUAUGGGGGAC','-')
```

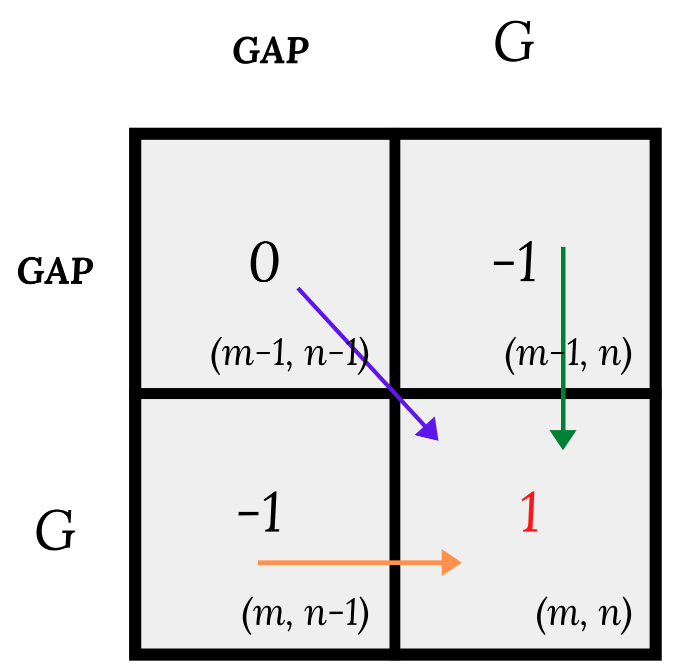
```
'Met-Gly-Asp'
```

def global_alignment_matrix(sequence_one,sequence_two)

Parameters: two sequences of nucleotides of type string.

The goal: I wanted to create a function called global_alignment_matrix that would make it much easier to globally align two sequences by performing the calculations necessary and returning a 2D matrix that one can use to figure out the optimal path.

The logic (in brief): There are four possible options in the Needleman-Wunsch algorithm, as demonstrated by the following 2x2 matrix. I used these calculations in my code - a snippet of which is shown to the right - on the matrix I created initially.



Options:

Vertical (m–1, n): -1 + (*gap*) = -2

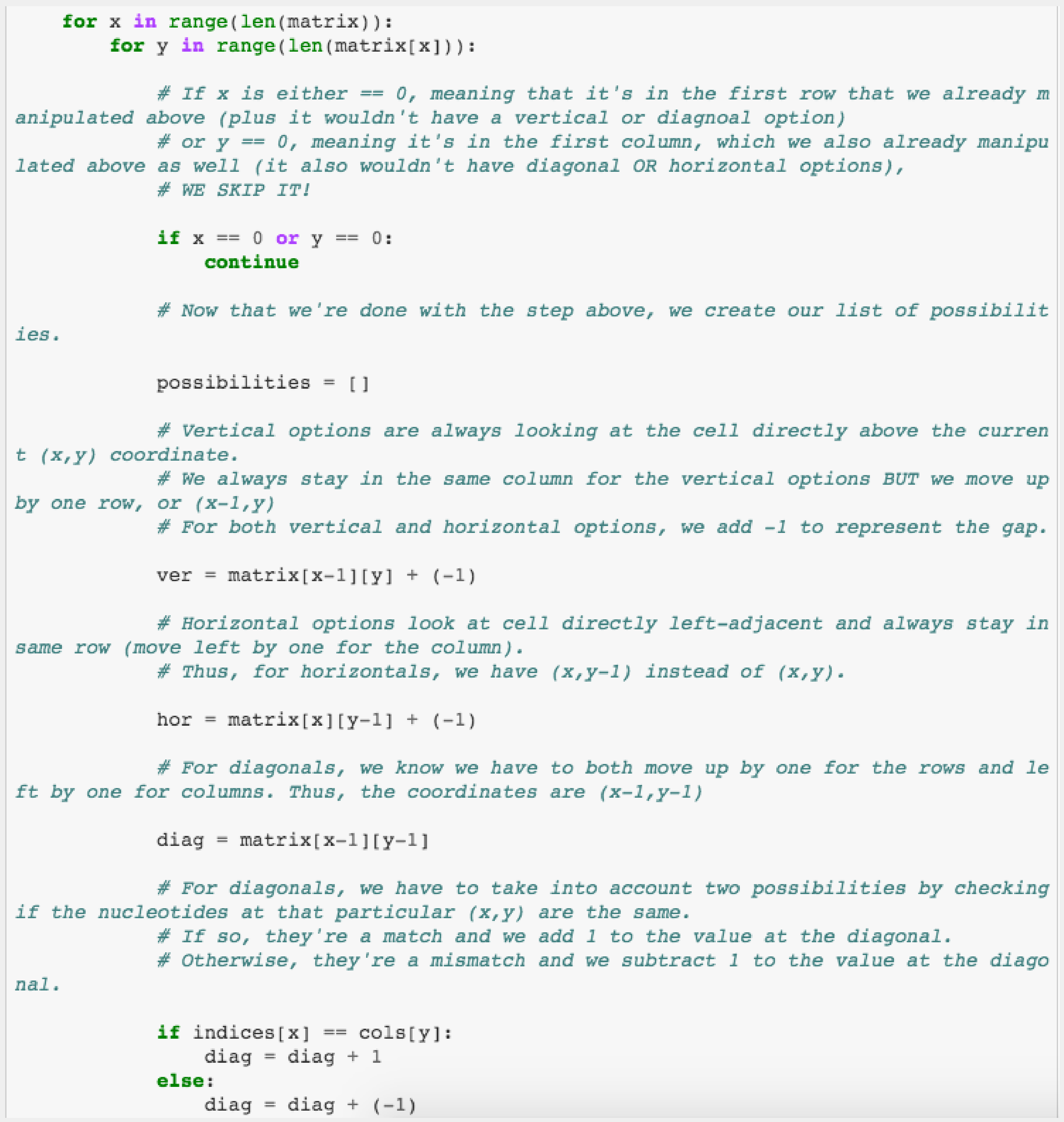
Horizontal (m, n-1): -1 + (*gap*) = -2

Diagonal (m-1, n-1): 0 + (*match*) = 1 { mismatch: -1, match: 1 }

Optimize:

max (-2, -2, 1) == 1

The Code (Partial)



```
    for x in range(len(matrix)):
        for y in range(len(matrix[x])):

            # If x is either == 0, meaning that it's in the first row that we already m
anipulated above (plus it wouldn't have a vertical or diagnoal option)
            # or y == 0, meaning it's in the first column, which we also already manipu
lated above as well (it also wouldn't have diagonal OR horizontal options),
            # WE SKIP IT!

            if x == 0 or y == 0:
                continue

            # Now that we're done with the step above, we create our list of possibilit
ies.

            possibilities = []

            # Vertical options are always looking at the cell directly above the curren
t (x,y) coordinate.
            # We always stay in the same column for the vertical options BUT we move up
by one row, or (x-1,y)
            # For both vertical and horizontal options, we add -1 to represent the gap.

            ver = matrix[x-1][y] + (-1)

            # Horizontal options look at cell directly left-adjacent and always stay in
same row (move left by one for the column).
            # Thus, for horizontals, we have (x,y-1) instead of (x,y).

            hor = matrix[x][y-1] + (-1)

            # For diagonals, we know we have to both move up by one for the rows and le
ft by one for columns. Thus, the coordinates are (x-1,y-1)

            diag = matrix[x-1][y-1]

            # For diagonals, we have to take into account two possibilities by checking
if the nucleotides at that particular (x,y) are the same.
            # If so, they're a match and we add 1 to the value at the diagonal.
            # Otherwise, they're a mismatch and we subtract 1 to the value at the diago
nal.

            if indices[x] == cols[y]:
                diag = diag + 1
            else:
                diag = diag + (-1)
```

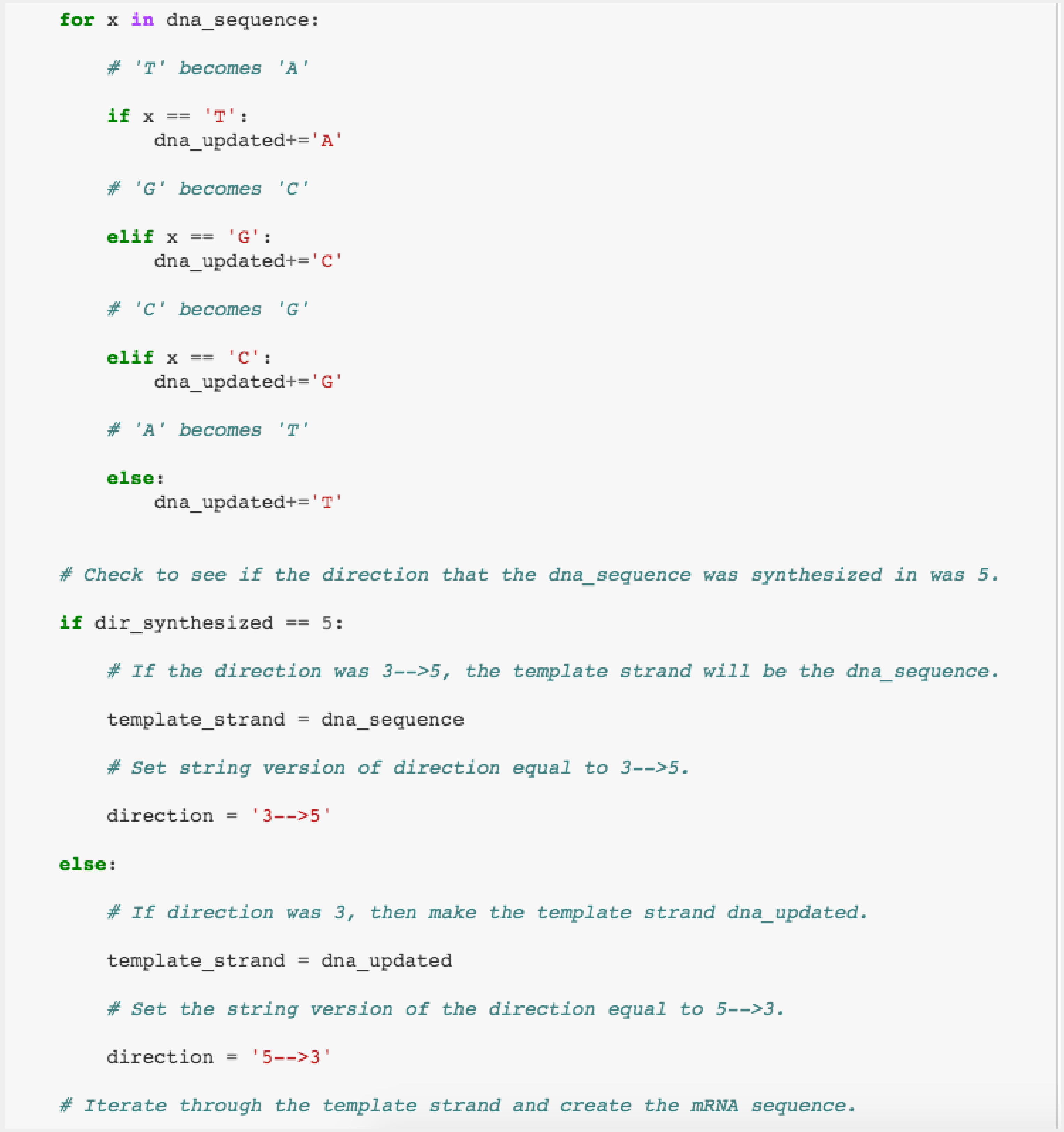
def transcription(dna_sequence,dir_synthesized)

Parameters: a sequence of nucleotides of type string and a direction of synthesis of type int - options for int are 3 | 5.

The goal: The aim of this function is to identify the template and coding strand based on the direction of synthesis passed in.

The logic (in brief): I first identified the complementary nucleotides in the dna_sequence. I then stored these nucleotides in a string called dna_updated. Next, I checked the value for dir_synthesized. If the dir_synthesized == 3, then the direction would be regarded as 5-->3. If the dir_synthesized == 5, then the direction would be regarded as 3-->5. Since mRNA is only synthesized in the 5-->3 direction, the template strand can only be one that runs from 3-->5. Therefore, if the sequence passed in was in the direction 3-->5, it is regarded as the template strand. Otherwise, the daughter DNA strand that is also formatted and manipulated in the function is regarded as the template strand.

The Code (Partial)



```
    for x in dna_sequence:

        # 'T' becomes 'A'

        if x == 'T':
            dna_updated+='A'

        # 'G' becomes 'C'

        elif x == 'G':
            dna_updated+='C'

        # 'C' becomes 'G'

        elif x == 'C':
            dna_updated+='G'

        # 'A' becomes 'T'

        else:
            dna_updated+='T'

    # Check to see if the direction that the dna_sequence was synthesized in was 5.

    if dir_synthesized == 5:

        # If the direction was 3-->5, the template strand will be the dna_sequence.

        template_strand = dna_sequence

        # Set string version of direction equal to 3-->5.

        direction = '3-->5'

    else:

        # If direction was 3, then make the template strand dna_updated.

        template_strand = dna_updated

        # Set the string version of the direction equal to 5-->3.

        direction = '5-->3'

    # Iterate through the template strand and create the mRNA sequence.
```

Some cool and curious cases that broke my code!

Mostly, it was with the matrix.

Initial failures:

I initially tried to emulate the large, 8x8 matrix and perform my calculations on it. But, I didn't entirely understand how to even begin with calculating my options for horizontal, vertical, and diagonal. Thus, the first thing I did was create an *algorithm* to address all of these different cases and solve my issue in steps. I also made the matrix much smaller, much like the matrix in the slide (leveraging the powerful concept of decomposition). This allowed me to manually calculate the new values and see what cool math was actually happening behind the scenes.

What broke my code:

I started tinkering with my code initially by bringing it outside of a function environment and just testing it with small sequences like 'ACG' and 'TAAA', for example. Once I had figured out the underlying computations, I tried to bring this code into a function. It immediately broke and returned an incorrect matrix! This was because I didn't go through as carefully as I needed to when reviewing all of the variables and loops that I had brought over to make sure all necessary updates were completed. Once I did that, all was fixed!

Sources

Needleman–Wunsch algorithm

Wikipedia article [mentioned, '8x8 matrix' (page 8) and description of global_alignment_matrix]
https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm#:~:text=The%20Needleman%E2%80%93Wunsch%20algorithm%20is,Needleman%20and%20Christian%20D.

Pictures

The Matrix image: https://screencrush.com/the-matrix-reloaded-15-anniversary-defense/