[DS4Bio] Coding: Python Coding of Genetic Coding

Data Science for Biology

Notebook developed by: *Sarp Dora Kurtoglu, Kinsey Long*
Supervised by: *Steven E. Brenner*

Learning Outcomes

In this notebook, you will review or learn about:

- String manipulation
- Building for loops for counting
- Building functions
- Using dictionaries
- Input validation

In this assignment, we will explore ways to write Python code for modeling transcription and translation.

Helpful Data Science Resources

Here are some resources you can check out while doing this notebook.

- Reference Sheet for the datascience Module
- Documentation for the datascience Module
- Khan Academy Overview of Transcription
- Khan Academy Overview of Translation

Peer Consulting

If you find yourself having trouble with any content in this notebook, Data Peer Consultants are an excellent resource. Click here to locate live help.

Peer Consultants are there to answer all data-related questions, whether it be about the content of this notebook, applications of data science in the world, or other data science courses offered at Berkeley.

To prepare our notebook environment, run the following cell which imports the necessary packages. It will print `All necessary packages have been imported.` below the cell when it's completed importing.

This notebook includes the same packages as the intro lab. It also include `re`, which stands for `regex`, a package often used for more sophisticated string operations.

In [1]:
```
# Run this cell
from datascience import *
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as sp
import re
plt.style.use('fivethirtyeight')
from IPython.display import Image
print("All necessary packages have been imported.")
```

```
All necessary packages have been imported.
```

Introduction

The Central Dogma

The **central dogma of molecular biology** is a *__theory__* stating that genetic information flows in one direction, from DNA --> RNA --> protein. This is predominantly true, but there are some exceptions. The conversion of information from DNA to RNA is called **transcription** and the conversion of information from RNA to protein is called **translation**.

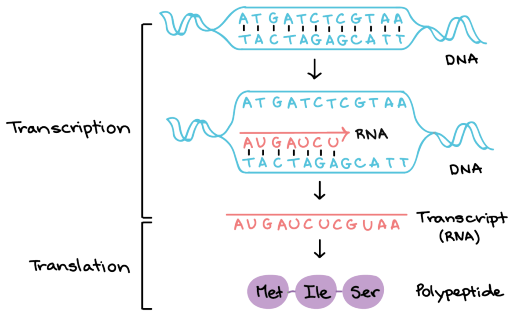
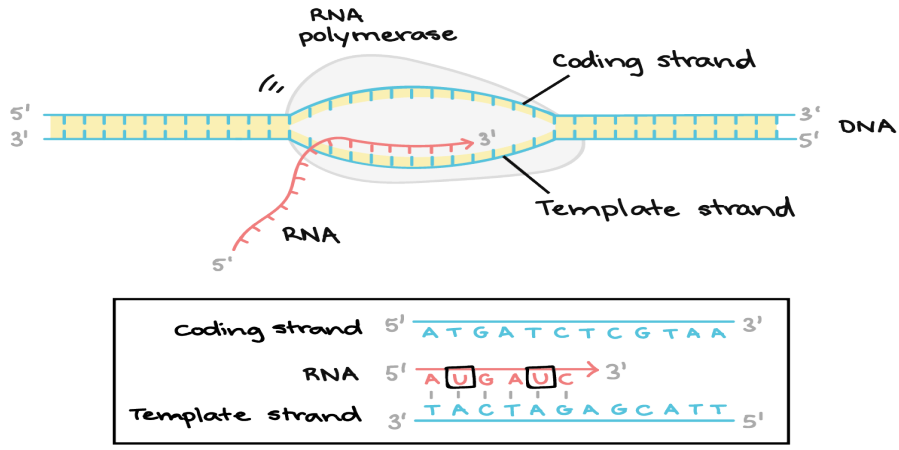


Image Source: Khan Academy

Transcription

In this lab, we will look at the DNA sequence of Green Fluorescept Protein (GFP). GFP is a naturally fluorescent protein that has become a prominent tool for biology researchers to visualize structures. The coding strand of GFP DNA (5' -> 3') can be sourced from NCBI.

A diagram of transcription is shown below:



Source: Khan Academy

During transcription, RNA polymerase reads in a 3' -> 5' direction on the template strand, synthesizing the new complementary mRNA in a 5' -> 3' direction. The template strand is the strand that is actually transcribed. The coding strand is the complementary DNA strand to the template strand. The resulting mRNA has a similar sequence to the coding strand, but with uracil instead of thymine.

Conventionally, DNA sequences are written in a 5' -> 3' direction and the coding strand is used to report the DNA sequence of a gene. In this assignment, we will start with the **5' -> 3' template strand** of GFP and computationally transcribe it into mRNA. Later, we will computationally translate that mRNA into the amino acid sequence.

Translation

DNA translation is a fundamental process in molecular biology, where mRNA is used as a guide to generate an amino acid sequence. Each set of three nucleotides on the mRNA, called a **codon**, corresponds to a specific amino acid, the building block of proteins. Transfer RNA (tRNA) molecules bring the appropriate amino acids to the ribosome, matching their anticodon sequences to the mRNA codons. The ribosome facilitates the formation of peptide bonds between these amino acids, sequentially building a polypeptide chain. This chain folds into a functional protein, determined by the original DNA sequence.

---

1. Template Strand of GFP

To begin, we will assign the 5' -> 3' template strand of GFP to the variable `template_strand`

In [2]:
```
#Just run this cell
template_strand = "atacactccagtagcctatttaataagaaaatagcccctattaataacatcaataaattat
template_strand
```

Out[2]:
```
'atacactccagtagcctatttaataagaaaatagcccctattaataacatcaataaattattcataaaatttttaatga
atctataaatatataaataaagtctcagcctgaatttaaccaggaaccctgagaatttagtaattgttcggacacttta
gtgtcaattggaagtctggacatttatttgtatagttcatccatgccatgtgtaatcccagcagctgttacaaactcaa
gaaggatcatgtgatctctcttttcgttgggatctttggaaagggcagattgtgtggacaggtaatggttgtctggtaa
aaggacagggccatcgccaattggagtattttgttgataatggtctgctaattgaacgcttccatctttaatgttgtgt
ctaattttgaagttaactttgattccattctttggtttgtctgccatgatgtatacattatgtgagttatagttgtatt
ccattttgtgtccaagaatgtttccatcttctttaaaatcaatacctttttaactcgattctattaacaagggtatcacc
ttcaaacttgacttcagcacgtgtcttgtagttcccgtcatctttgtaaaatatagttctttcctgtacataaccttcg
ggcatggcactcttgaaaaagtcatgctgtttcatatgatctgggtatcttgaaaagcattgaacaccataagagaaag
tagtgacaagtgttggccatggaacaggtagcttcccagtagtgcaaataaatttaagggtaagttttccgtatgttgc
atcaccttcacccctctccactgacagagaatttttgcccattaacatcgccatctaattcaacaagaattgggacaact
ccagtgaaaagttcttctcctttactcatctttgttatcttttattcgtgtgta'
```

QUESTION 1: How many nucleotides long is the GFP template strand? (1 point)

> *Hint:*
> *- The built-in function* `len(__)` *could be useful.*

- Use the code cell below.

In [3]:
```
# Question 1 -- YOUR CODE HERE
gfp_length = len(template_strand)

print(f"The GFP sequence is {gfp_length} nucleotides long.")
```

```
The GFP sequence is 922 nucleotides long.
```

QUESTION 2a: What is the percentage of adenosine (a) nucleotides in the GFP template strand? How about guanine? Thymine? Cytosine? Fill in the skeleton code below. (4 points)

> *Hint:*
> *- Think about iteration and how we can use it throught the specific variable we want.*

In [4]:
```
# Question 2a -- YOUR CODE HERE
# Intialize variables that will represent the counts of each nucleotide.
adenosine = 0
thymine = 0
guanine = 0
cytosine = 0

# Iterate through the template strand string using a for loop and conditional st
for character in template_strand:
  if character == 'a':
    adenosine += 1
  elif character == 't':
    thymine += 1
  elif character == 'g':
    guanine += 1
  elif character == 'c':
    cytosine += 1

#Let's get the length of the string to calculate the percent
length = len(template_strand)
```

```
#Use the nucleotide counts and the total length of the sequence to calculate the
percentage_adenosine = adenosine / length * 100
percentage_thymine = thymine / length * 100
percentage_guanine = guanine / length * 100
percentage_cytosine = cytosine / length * 100

print(f"The sequence is {percentage_adenosine}% A")
print(f"The sequence is {percentage_thymine}% T")
print(f"The sequence is {percentage_guanine}% G")
print(f"The sequence is {percentage_cytosine}% C")
```

```
The sequence is 28.850325379609544% A
The sequence is 34.924078091106296% T
The sequence is 17.787418655097614% G
The sequence is 18.43817787418655% C
```

Notice how the percentages you calculated in the previous equation show a large number of seemingly significant figures. Considering the length of our GFP sequence, this is not an appropriate number of significant figures to report the nucleotide frequencies.

QUESTION 2b: Use `np.round(float, number_of_decimal_places)` to round each of the percentages calculated in Question 2a to an appropriate number of decimal places. (2 points)

> - The new percentages should have the same number of significant figures as the length of the GFP sequence.
> - You must use the variables `percentage_adenosine`, `percentage_thymine`, `percentage_guanine`, and `percentage_cytosine`, instead of copy and pasting the float values directly.

In [5]:
```
# YOUR CODE HERE
percentage_adenosine_sigfig = round(percentage_adenosine, 2)
percentage_thymine_sigfig = round(percentage_thymine, 2)
percentage_guanine_sigfig = round(percentage_guanine, 2)
percentage_cytosine_sigfig = round(percentage_cytosine, 2)

#DO NOT CHANGE THIS CODE
print(f"The sequence is {percentage_adenosine_sigfig}% A")
print(f"The sequence is {percentage_thymine_sigfig}% T")
print(f"The sequence is {percentage_guanine_sigfig}% G")
print(f"The sequence is {percentage_cytosine_sigfig}% C")

print(f"Sanity check sum: {percentage_adenosine_sigfig + percentage_thymine_sigf
```

```
The sequence is 28.85% A
The sequence is 34.92% T
The sequence is 17.79% G
The sequence is 18.44% C
Sanity check sum: 100.0
```

The GC content of DNA is the proportion of guanine and cytosine bases as a fraction of all bases. It is an important statistic for several reasons. The GC content of DNA influences the stability of the DNA molecule, as G-C pairs form stronger bonds than A-T pairs, leading to greater stability in high GC regions. This impacts DNA's melting

temperature, replication, and transcription, especially since high GC areas can form complex structures. Moreover, GC content varies across different organisms and affects gene regulation (in both DNA and RNA) and expression. In biotechnological applications, such as PCR primer design, GC content is vital for determining specificity and efficiency. It is also used to understand horizontal gene transfer.

QUESTION 3: Build a function `gc_content` that calculates the GC content (as a percentage) of an input sequence. It should report the GC content to 1 decimal place. Then, call the function to calculate the GC content of the GFP template strand. (3 points)

> *Hint:*
> *- Feel free to use an approach similar to Question 2, or your own approach.*

In [6]:
```
# Question 3 -- YOUR CODE HERE
def gc_content(sequence):
    g_content = sequence.count('g')
    c_content = sequence.count('c')
    length = len(sequence)
    gc_content = (g_content + c_content) / length * 100
    return np.round(gc_content, 1)

gc_content_gfp = gc_content(template_strand)

print(f"The GC content of GFP is {gc_content_gfp}%.")
```

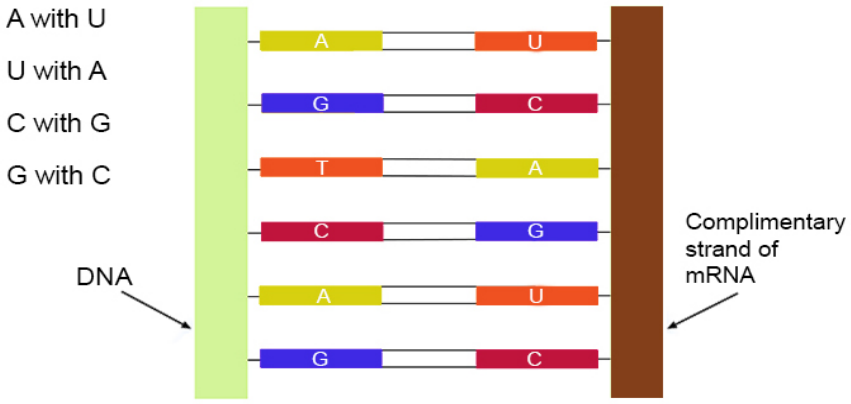
```
The GC content of GFP is 36.2%.
```

---

2. Transcription

QUESTION 4a: Now, we would like to computationally transcribe the GFP template strand into mRNA. First, create a dictionary called `base_pairs` to pair the 4 DNA nucleotides with their complementary RNA nucleotides for transcription purposes. (2 points)

> - Hint: Keep in mind that RNA does not have exactly the same nucleotide bases as DNA.



In [7]:
```
# Question 4a -- YOUR CODE HERE

base_pairs = {
    "g": "c" ,
    "c": "g" ,
    "a": "t" ,
    "t": "a" ,
}
```

QUESTION 4b: Now, write a function `transcribe()` that takes a DNA template strand sequence (**5'-> 3'**) and outputs the resulting mRNA transcript (**5'->3'**) that is produced. (5 points)

> Hint:

- **Strand direction is very important in this question.** What is the directional relationship between the template strand and mRNA transcript?
- There are a many possible approaches to this question. In one approach, you could try implementing a for loop and the dictionary method `.get()`. Feel free to use your own approach and as many lines of code as you need.
- You can check your work by looking at the printed result of `q4b_check`, which transcribes an input sequence of `'atcg'`. What do you expect it to output if your function works correctly?

In [8]:
```
# Question 4b -- YOUR CODE HERE
def transcribe(seq):
    # Initialize an empty string to store the transcribed sequence
    transcribed_seq = ""

    # Iterate through the input sequence and use the base_pairs
    for nucleotide in seq:
        transcribed_seq += base_pairs[nucleotide]

    # For each t nucleotide in the transcribe sequence, replace it with a u
    transcribed_seq = transcribed_seq.replace("t", "u")
    reversed_transcribed_seq = transcribed_seq[::-1]
    return reversed_transcribed_seq

#DO NOT CHANGE THIS CODE - this is for you to check that your function is workin
q4b_check = transcribe("atcg")
print(q4b_check)
```

```
cgau
```

Converting from template strand (5' -> 3') to mRNA transcript (5' -> 3')

Given a template strand "atcg" in the 5' to 3' direction, we first get the complement. The complement is "tagc". In mRNA, uracil (U) replaces thymine (T), so we convert all T's to U's, resulting in "uagc".

However, mRNA is synthesized in the 5' to 3' direction, so we reverse the entire string to get "cgau". Why? The directional relationship between the template strand and the mRNA is that they are antiparallel:

In the mRNA transcription process, the "template strand" refers to the strand of DNA that RNA polymerase directly reads and uses as a template to build the complementary mRNA molecule. The template strand runs in the 3' to 5' direction, allowing the new mRNA strand to be synthesized in the 5' to 3' direction. Essentially, the template strand provides the sequence information for the mRNA transcript.

Detailed Steps:

Complementary Base Pairing: The DNA template strand is read in the 3' to 5' direction by RNA polymerase. The enzyme synthesizes the mRNA strand in the 5' to 3' direction by adding complementary RNA nucleotides. For example, if the DNA template strand has an adenine (A), the RNA polymerase adds a uracil (U) to the mRNA strand. Uracil Replacement: In RNA, uracil (U) replaces thymine (T). So, wherever there is a thymine in the DNA template, it is replaced by uracil in the mRNA. Antiparallel Nature: The mRNA strand is antiparallel to the DNA template strand. This means that if the DNA template strand is read from 3' to 5', the mRNA strand is synthesized from 5' to 3'.

Reversing the Sequence: Since the template strand is given in the 5' to 3' direction, we need to reverse it to get the correct mRNA sequence in the 5' to 3' direction. By following these steps, we ensure that the mRNA transcript accurately reflects the genetic information encoded in the DNA template strand.

QUESTION 5: Now let's put to use the `transcribe` function we just defined in Question 4. We want to obtain the mRNA sequence (5' -> 3') from the `template_strand` (5' -> 3') of the GFP gene. Save the final mRNA sequence as a string called `mrna`. (1 point)

In [9]:
```
# Question 5 -- YOUR CODE HERE

mrna = transcribe(template_strand)
mrna
```

Out[9]:
```
'uacacacgaauaaaagauaacaaagaugaguaaaggagaagaacuuuucacuggaguugucccaauucuuguugaauu
agauggcgauguuaaugggcaaaaauucucugucaguggagagggugaaggugaugcaacauacggaaaacuuacccuu
aaauuuauuugcacuacugggaagcuaccuguuccauggccaacacuugucacuacuuucucuuaugguguucaaugcu
uuucaagauacccagaucauaugaaacagcaugacuuuuucaagagugccaugcccgaagguuauguacaggaaagaac
uauauuuuacaaagaugacgggaacuacaagacacgugcugaagucaaguuugaaggugauacccuuguuaaauagaauc
gaguuaaaagguauugauuuuaaagaagauggaaacauucuuggacacaaaauggaauacaacuauaacucacauaaug
uauacaucauggcagacaaaccaaagaauggaaucaaaguuaacuucaaaauuagacacaacauuaaagauggaagcgu
ucaauuagcagaccauuaucaacaaaauacuccaauuggcgauggcccuguccuuuuaccagacaaccauuaccugucc
acacaaucugcccuuuccaaagaucccaacgaaaagagagaucacaugauccuucuugaguuuguaacagcugcuggga
uuacacauggcauggaugaacuauacaaauaaauguccagacuuccaauugacacuaaaguguccgaacaauuacuaaa
uucucaggguuccuggguuaaauucaggcugagacuuuauuuauauauuuuauagauucauuaaaaauuuuaugaauaauuu
auugauguuauuaauaggggcuauuuucuuauuaaaauaggcuacuggaguguau'
```

---

3. Open Reading Frame

An Open Reading Frame (ORF) is a continuous stretch of DNA that begins with a start codon and ends with a stop codon. This sequence of DNA has the potential to be

translated into a protein. ORFs are significant in molecular biology and genetics because they represent regions of a gene that can encode a protein. The identification of ORFs is crucial in genomic studies and bioinformatics, as it helps in predicting the presence and structure of genes in a DNA sequence. The presence of an ORF in a DNA sequence does not guarantee that it will be expressed into a functional protein, but it is a strong indicator of a gene's coding potential.

In the case of GFP, the first ORF is the one that codes for the GFP protein. DNA may have multiple ORFs.

QUESTION 6a: For this question, we will take a step-by-step approach to find the first open reading frame (ORF) of our transcript. First, fill in the code below to assign appropriate codons to `start_codon` and `stop_codons`. (2 points)

> Hint:
> - `start_codon` should be one codon, as a string. `stop_codons` should be a list of three codon strings.
> - Make sure your codons are written in lowercase.

In [10]:
```
#Question 6a -- YOUR CODE HERE
start_codon = "aug"
stop_codons = ["uaa", "uag", "uga"]
```

What are codons? AUG is the universal start codon in mRNA translation and it signals the beginning of translation and codes for the amino acid methionine (Met). This is the codon that the ribosome recognizes to start the protein synthesis

stop codons signal the termination of translation in mRNA. There are 3 stop codons:

1. UAA
2. UAG
3. UGA

stop codons do not code for any amino acid so when the ribosome encounters one of them it stops translation

QUESTION 6b: Assign the index of the first RNA nucleotide in the first start codon in the mRNA transcript to `start_codon_index`. (2 points)

> Hint:
> - Consider using the `.find` string method.
> - For example, if our mRNA transcript was "ggaugcg", `start_codon_index` should be 2.

In [11]:
```
# Question 6b -- YOUR CODE HERE
start_codon_index = mrna.find(start_codon)
start_codon_index
```

Out[11]: 25

QUESTION 6c: Use Python string splicing to remove the 5' UTR of the mRNA transcript. The UTR is the untranscribed region before the first coding nucleotides. The variable `mrna_start` should be a string that begins with the start codon and includes the remainder of the `mrna` transcript. (2 points)

In [12]:
```
#Question 6c -- YOUR CODE HERE
spliced_mrna = mrna[start_codon_index:]
spliced_mrna
```

Out[12]:
```
'augaguaaaggagaagaacuuuucacuggaguugucccaauucuuguugaauuagauggcgauguuaaugggcaaaaa
uucucugucaguggagagggugaaggugaugcaacauacggaaaacuuacccuuaaauuuauuugcacuacugggaagc
uaccuguuccauggccaacacuugucacuacuuucucuuaugguguucaaugcuuuucaagauacccagaucauaugaa
acagcaugacuuuuucaagagugccaugcccgaagguuauguacaggaaagaacuauauuuuacaaagaugacgggaac
uacaagacacgugcugaagucaaguuugaaggugauacccuuguuaauagaaucgaguuaaaagguauugauuuuaaag
aagauggaaacauucuuggacacaaaauggaauacaacuauaacucacauaauguauacaucauggcagacaaaccaaa
gaauggaaucaaaguuaacuucaaaauuagacacaacauuaaagauggaagcguucaauuagcagaccauuaucaacaa
aauacuccaauuggcgauggcccuguccuuuuaccagacaaccauuaccuguccacacaaucugcccuuuccaaagauc
ccaacgaaaagagagaucacaugauccuucuugaguuuguaacagcugcugggauuacacauggcauggaugaacuaua
caaauaaauguccagacuuccaauugacacuaaaguguccgaacaauuacuaaauucucaggguuccugguuaaauuca
ggcugagacuuuauuuauauauuuauagauucauuaaaauuuuaugaauaauuuauugauguuauuaauaggggcuauu
uucuuauuaaauaggcuacuggaguguau'
```

The Untranslated Regions (UTRs) are sections of an mRNA transcript that are not translated into protein and they play crucial regulatory roles in gene expression. There are two types of UTRs:

1. 5' UTR (Five-Prime Untranslated Region)
    - This is the region before the start codon ("AUG")
    - Located at the beginning (5' end) of the mRNA
    - regulates translation efficiency which then in turn affectrs how well the ribosome binds to the mRNA
    - in prokaryotes, it also contains this sequence called the Shine-Dalgarno sequencve whcih helps ribosome bind to the mRNA
    - in eukaryotes, the Kozak sequence near the start codon helps translation initiation
2. 3' UTR (Three-Prime Untranslated Region)
    - region after the stop codon
    - it is located at the end (3' end) of the mRNA
    - helps mRNA stability, translation regulation and localization

Overall UTRs help with:

- Regulate Translation efficiency -> How much protein is produced
- mRNA Stability -> the 3' UTR especially the poly(A) tail protects the mRNA from being degraded too quickly
- Localization -> Some 3' UTR sequences help direct the mRNA to specific parts of the cell

QUESTION 6d: In Question 6b and 6c, we identified the first start codon in the mRNA transcript and removed the 5' UTR of the transcript so that it begins at the start codon.

However, to find the terminal end of the transcript, we cannot simply search for a stop codon anywhere in the sequence and remove the 3' UTR of the transcript by removing all sequences after the stop codon. Provide two reasons why repeating the same coding approach would not work. (2 points)

Reason 1: In a mRNA sequence, stop codons can appear randomly in different reading frames outside of our actual mRNA transcript. If we simply search for the first occurence of a stop codon anywhere in the sequence, we might mistakenly "cut" the mRNA too early before the actual protein-coding region is fully translated ultimately leading to a truncated protein that lacks essential functional domains. As a side note, these "premature" stop codons exists due to mutations or even sequencing errors.

Reason 2: Translation occurs in sets of codons starting from the first start codon (AUG). Basically, the correct stop codon must be in the same reading frame as this start codon. If we scan for stop codons without ensuring they align with the same reading frame as the start codon, we might accidentaly select a stop codon that appears in different reading frame. Ultimately this too leads to a premature termination of translation or even too late.

QUESTION 6e: Build a function called `codon_split` that converts a string into a list of triplet codons (as strings). Then, use the function to convert `mrna_start` into a list of codons, and assign it to the variable `codons_orf`. (5 points)

> - Any leftover sequence that cannot form a complete codon should be excluded.
> - For example, `codon_split("acgcgagca")` should output `["acg", "cga", "gca"]` and `codon_split("acgcgagc")` should output `["acg", "cga"]`.
> - There are many different ways to approach this problem. You do not have to follow the skeleton code if you prefer to use another method to achieve the same result.

In [13]:
```
# Question 6d -- YOUR CODE HERE

def codon_split(sequence):
    codons = []
    for i in range(0, len(sequence), 3):
        codon = sequence[i:i+3]
        if len(codon) < 3:
            break
        codons.append(codon)
    return codons

codons_orf = codon_split(spliced_mrna)

#DO NOT CHANGE THIS CODE
print(codon_split("acgcgagca"))
print(codon_split("acgcgagc"))
try:
    print(codons_orf[len(codons_orf)//3]) #GRADER TEST
```

```
except:
    print("N/A")
```

```
['acg', 'cga', 'gca']
['acg', 'cga']
uac
```

QUESTION 6f: Assign the Open Reading Frame of the mrna as a string to `mrna_frame`. The template skeleton approach below involves adding string codons to `mrna_frame` until a stop codon is read. The final transcript should include both the start and stop codon. You do not have to follow the skeleton code if you have an alternative approach. (4 points)

In [14]:
```
# Question 6e -- YOUR CODE HERE

mrna_frame = ""

# this might be problematic I think if codons_orf is empty and will error I beli
for codon in codons_orf:
    mrna_frame += codon
    if codon in stop_codons:
        break     #break causes the code to exit the for loop

#DO NOT CHANGE THIS CODE
print(mrna_frame)
print()
print(mrna_frame[:3], mrna_frame[-3:])
print()
print(mrna_frame[len(mrna_frame)//5: len(mrna_frame)//5 + 5]) #GRADER TEST
```

```
augaguaaaggagaagaacuuuucacuggaguugucccaauucuuguugaauuagauggcgauguuaaugggcaaaaauuc
ucugucaguggagagggugaaggugaugcaacauacggaaaacuuacccuuaaauuuauuugcacuacugggaagcuaccu
guuccauggccaacacuugucacuacuuucucuuauggugguucaaugcuuuucaagauacccagaucauaugaaacagcau
gacuuuuucaagagugccaugcccgaagguuauguacaggaaagaacuauauuuuacaaagaugacgggaacuacaagaca
cgugcugaagucaaguuugaaggugauacccuuguuaauagaaucgaguuaaaagguauugauuuuaaagaagauggaaac
auucuuggacacaaaauggaauacaacuauaacucacauaauguauacaucauggcagacaaaccaaagaauggaaucaaa
guuaacuucaaaauuagacacaacauuaaagauggaagcguucaauuagcagaccauuaucaacaaaauacuccaauuggc
gauggcccuguccuuuuaccagacaaccauuaccuguccacacaaucugcccuuuccaaagaucccaacgaaaagagagau
cacaugauccuucuugaguuuguaacagcugcugggauuacacauggcauggaugaacuauacaaauaa

aug uaa

cacua
```

QUESTION 6g: Finally, combine your code from Question 6b - 6g to create the function `orf(seq)`. It should take in a 5' -> 3' mRNA sequence in upper or lower case and then return the corresponding Open Reading Frame (including the start and stop codons) of the sequence in lowercase letters. In addition, it should be able to accept a sequence of uppercase or lowercase nucleotides. (4 points)

> Hint:
> - You can use `string.upper()` to convert every character in a string to upper case, or `string.lower()` to convert every character in a string to lowercase.

In [15]:
```
# Question 6g - YOUR CODE HERE
def orf(seq):
    seq = seq.lower()
    start_codon = "aug"
    stop_codons = ["uaa", "uag", "uga"]

    start_codon_index = seq.find(start_codon)
    spliced_seq = seq[start_codon_index:]

    codons = codon_split(spliced_seq)
    orf_seq = ""

    for codon in codons:
        orf_seq += codon
        if codon in stop_codons:
            break

    return orf_seq


#DO NOT CHANGE THIS CODE
orf_mrna = orf(mrna)
print("Do orf(mrna) and mrna_frame output the same ORF? ", orf_mrna == mrna_fram
print("Does orf accept both uppercase and lowercase? ", orf(mrna.upper()) == orf
print("Does orf return a lowercase string?", orf_mrna.islower())
```

```
Do orf(mrna) and mrna_frame output the same ORF?  True
Does orf accept both uppercase and lowercase?  True
Does orf return a lowercase string? True
```

---

4. Translation

Finally, for the last part of the central dogma, we will translate our mRNA into an amino acid sequence.

The standard "universal" codon translation table is in the dictionary below. The keys of this dictionary are the 3-nucleotides long codons and corresponding values are the one-letter symbols of the amino acids they code for. Notice that the dictionary is in uppercase.

Be sure to **run the cell below** so the `codon_table` variable gets defined.

In [16]:
```
codon_table = {
    'AUA':'I', 'AUC':'I', 'AUU':'I', 'AUG':'M',
    'ACA':'T', 'ACC':'T', 'ACG':'T', 'ACU':'T',
    'AAC':'N', 'AAU':'N', 'AAA':'K', 'AAG':'K',
    'AGC':'S', 'AGU':'S', 'AGA':'R', 'AGG':'R',
    'CUA':'L', 'CUC':'L', 'CUG':'L', 'CUU':'L',
    'CCA':'P', 'CCC':'P', 'CCG':'P', 'CCU':'P',
    'CAC':'H', 'CAU':'H', 'CAA':'Q', 'CAG':'Q',
    'CGA':'R', 'CGC':'R', 'CGG':'R', 'CGU':'R',
    'GUA':'V', 'GUC':'V', 'GUG':'V', 'GUU':'V',
    'GCA':'A', 'GCC':'A', 'GCG':'A', 'GCU':'A',
    'GAC':'D', 'GAU':'D', 'GAA':'E', 'GAG':'E',
    'GGA':'G', 'GGC':'G', 'GGG':'G', 'GGU':'G',
```

```
    'UCA':'S', 'UCC':'S', 'UCG':'S', 'UCU':'S',
    'UUC':'F', 'UUU':'F', 'UUA':'L', 'UUG':'L',
    'UAC':'Y', 'UAU':'Y', 'UAA':'', 'UAG':'',
    'UGC':'C', 'UGU':'C', 'UGA':'', 'UGG':'W'}
```

QUESTION 7: Use the previous `codon_split` function to convert the mRNA open reading frame into a list of **uppercase** codon strings. (2 points)

> *Hint:*
> *- You can use `string.upper()` to convert each character of the string to upper case.*

In [17]:
```
# Question 7 -- YOUR CODE HERE

# Idk if I convoluted this way more than it should be but I get the feeling I di
codons = list(map(lambda x: x.upper(), codon_split(orf_mrna)))
print(codons)
```

```
['AUG', 'AGU', 'AAA', 'GGA', 'GAA', 'GAA', 'CUU', 'UUC', 'ACU', 'GGA', 'GUU', 'GU
C', 'CCA', 'AUU', 'CUU', 'GUU', 'GAA', 'UUA', 'GAU', 'GGC', 'GAU', 'GUU', 'AAU',
'GGG', 'CAA', 'AAA', 'UUC', 'UCU', 'GUC', 'AGU', 'GGA', 'GAG', 'GGU', 'GAA', 'GG
U', 'GAU', 'GCA', 'ACA', 'UAC', 'GGA', 'AAA', 'CUU', 'ACC', 'CUU', 'AAA', 'UUU',
'AUU', 'UGC', 'ACU', 'ACU', 'GGG', 'AAG', 'CUA', 'CCU', 'GUU', 'CCA', 'UGG', 'CC
A', 'ACA', 'CUU', 'GUC', 'ACU', 'ACU', 'UUC', 'UCU', 'UAU', 'GGU', 'GUU', 'CAA',
'UGC', 'UUU', 'UCA', 'AGA', 'UAC', 'CCA', 'GAU', 'CAU', 'AUG', 'AAA', 'CAG', 'CA
U', 'GAC', 'UUU', 'UUC', 'AAG', 'AGU', 'GCC', 'AUG', 'CCC', 'GAA', 'GGU', 'UAU',
'GUA', 'CAG', 'GAA', 'AGA', 'ACU', 'AUA', 'UUU', 'UAC', 'AAA', 'GAU', 'GAC', 'GG
G', 'AAC', 'UAC', 'AAG', 'ACA', 'CGU', 'GCU', 'GAA', 'GUC', 'AAG', 'UUU', 'GAA',
'GGU', 'GAU', 'ACC', 'CUU', 'GUU', 'AAU', 'AGA', 'AUC', 'GAG', 'UUA', 'AAA', 'GG
U', 'AUU', 'GAU', 'UUU', 'AAA', 'GAA', 'GAU', 'GGA', 'AAC', 'AUU', 'CUU', 'GGA',
'CAC', 'AAA', 'AUG', 'GAA', 'UAC', 'AAC', 'UAU', 'AAC', 'UCA', 'CAU', 'AAU', 'GU
A', 'UAC', 'AUC', 'AUG', 'GCA', 'GAC', 'AAA', 'CCA', 'AAG', 'AAU', 'GGA', 'AUC',
'AAA', 'GUU', 'AAC', 'UUC', 'AAA', 'AUU', 'AGA', 'CAC', 'AAC', 'AUU', 'AAA', 'GA
U', 'GGA', 'AGC', 'GUU', 'CAA', 'UUA', 'GCA', 'GAC', 'CAU', 'UAU', 'CAA', 'CAA',
'AAU', 'ACU', 'CCA', 'AUU', 'GGC', 'GAU', 'GGC', 'CCU', 'GUC', 'CUU', 'UUA', 'CC
A', 'GAC', 'AAC', 'CAU', 'UAC', 'CUG', 'UCC', 'ACA', 'CAA', 'UCU', 'GCC', 'CUU',
'UCC', 'AAA', 'GAU', 'CCC', 'AAC', 'GAA', 'AAG', 'AGA', 'GAU', 'CAC', 'AUG', 'AU
C', 'CUU', 'CUU', 'GAG', 'UUU', 'GUA', 'ACA', 'GCU', 'GCU', 'GGG', 'AUU', 'ACA',
'CAU', 'GGC', 'AUG', 'GAU', 'GAA', 'CUA', 'UAC', 'AAA', 'UAA']
```

QUESTION 8a: Using the `codon_table` dictionary defined for you above Question 7, write a function `translate()` that takes in a (5'->3') string, `mrna`, and a codon `table` as inputs, and outputs the corresponding sequence of amino acids after translation in string form. There should not be any amino acid symbol for the stop codon. (5 points)

In [18]:
```
# Question 8 -- YOUR CODE HERE
def translate(mrna, table):
    protein = ""

    # Ensure the mRNA is uppercase for consistency
    mrna = mrna.upper()

    # Split the mRNA sequence into codons
    codons = codon_split(mrna)

    # then we iterate through the codons and translate them into amino acids
```

```
    for codon in codons:
        if codon in table:
            amino_acid = table[codon]
            if amino_acid == "":
                break
            protein += amino_acid

    return protein
```

In [19]:
```
#DO NOT CHANGE THIS CELL, RUN THIS CELL TO CHECK YOUR CODE
def q8a_check():
    try:
        if translate("augggaucggugacgggguaa", codon_table) == "MGSVTG" and trans
            print("Q8a Test Case: Passed!")
        else:
            print("Q8b Test Case 1: Failed. Your function is not translating cor
    except:
        print("Q8b Test Case 1: Failed. Your function is not translating correct
q8a_check()
```

```
Q8a Test Case: Passed!
```

QUESTION 8b: It is also good coding practice to include input validation for functions. Create a function `valid_mrna_check(input)`, which will output `True` if the input is a translatable mRNA strand, and `False` if it is not. Then, incorporate `valid_mrna_check` function into `translate_updated`, which will translate the mRNA if it is valid. If the mRNA is invalid, it will output `"Error: Not Valid mRNA."` (6 points)

> Hint:
>
> - How long should a translatable mRNA sequence be?
> - What characters are in an mRNA sequence?
> - What should a translatable mRNA sequence start with and end with?

Valid mRNA Conditions

1. Length: It should be a multiple of 3 since we analyze in triplets of nucleotides (codons). Finally we start with a triplet, build in triplets and end with a triplet.
2. every mRNA sequence contains the nucleotides A,U,G,C
3. Every translatable mRNA must start with the start codon AUG and end with one of the stop codon [UAA, UAG, UGA]

In [20]:
```
# YOUR CODE HERE
def valid_mrna_check(string):
    string = str(string).upper()
    # check if the string contains valid nucleotides
    valid_nucleotides = set("ACGU")
    if not set(string).issubset(valid_nucleotides):
        return False

    # check if the string length is a multiple of 3
    if len(string) % 3 != 0:
        return False
```

```
    # Check if the start codon is AUG
    if string[:3] != "AUG":
        return False

    # Check if seq ends wuith stop codon
    stop_codons = ["UAA", "UAG", "UGA"]

    if string[-3:] not in stop_codons:
        return False

    return True



def translate_updated(mrna, table):
    if not valid_mrna_check(mrna):
        return "Error: Not Valid mRNA"
    else:
        return translate(mrna, table)
```

In [21]:
```
valid_mrna_check(12234)
```

Out[21]: False

In [22]:
```
#DO NOT CHANGE THIS CELL, RUN THIS CELL TO CHECK YOUR CODE
def q8b_check():
    testcase1 = "Q8b Test Case 1: Failed. What characters are in an mRNA sequenc
    testcase2 = "Q8b Test Case 2: Failed. What should a translatable mRNA sequen
    testcase3 = "Q8b Test Case 3: Failed. How long should a translatable mRNA se
    testcase4 = "Q8b Test Case 4: Failed. Your function/s are not recognizing va
    try:
        print("Q8b Test Case 1: Passed.") if not any([valid_mrna_check("applebot
    except:
        print(testcase1)
    try:
        print("Q8b Test Case 2: Passed.") if not any([valid_mrna_check("aggacgua
    except:
        print(testcase2)
    try:
        print("Q8b Test Case 3: Passed.") if not any([valid_mrna_check("augcuag"
    except:
        print(testcase3)
    try:
        print("Q8b Test Case 4: Passed.") if all([valid_mrna_check("auggcaggagua
    except:
        print(testcase4)

q8b_check()
```

```
Q8b Test Case 1: Passed.
Q8b Test Case 2: Passed.
Q8b Test Case 3: Passed.
Q8b Test Case 4: Passed.
```

QUESTION 9: Determine the amino acid sequence from the GFP mRNA. Save the final amino acid sequence as a string to the variable `aa_seq`. (1 point)

In [23]:
```
# Question 9 -- YOUR CODE HERE
aa_seq = translate_updated(orf_mrna, codon_table)
```

```
aa_seq
```

Out[23]:
```
'MSKGEELFTGVVPILVELDGDVNGQKFSVSGEGEGDATYGKLTLKFICTTGKLPVPWPTLVTTFSYGVQCFSRYPDHM
KQHDFFKSAMPEGYVQERTIFYKDDGNYKTRAEVKFEGDTLVNRIELKGIDFKEDGNILGHKMEYNYNSHNVYIMADKP
KNGIKVNFKIRHNIKDGSVQLADHYQQNTPIGDGPVLLPDNHYLSTQSALSKDPNEKRDHMILLEFVTAAGITHGMDEL
YK'
```

QUESTION 10: Finally, combine the transcription and translation code you have written throughout this assignment to build the function `translate_from_template(template_sequence)`. It should accept any 5' -> 3' DNA template strand, transcribe it into mRNA, determine the open reading frame, and then translate the mRNA into an amino acid sequence. (3 points)

In [24]:
```
# Question 9 -- YOUR CODE HERE
def translate_from_template(template_sequence):
    mrna_seq = transcribe(template_sequence)
    mrna_orf = orf(mrna_seq)
    aa_seq = translate_updated(mrna_orf, codon_table)
    return aa_seq

#DO NOT CHANGE THIS CODE
print("Do translate_from_template(template_strand) and aa_seq output the same se
print(translate_from_template("ttacgattcatacgtcgactccgtcat")[1:])
```

```
Do translate_from_template(template_strand) and aa_seq output the same sequence?
True
TESTYES
```

Congratulations! You have finished the coding lab! Don't forget you're also assigned the reversal lab this week and to check your PDF exports!
