

Codon Reversal: Reversing Amino Acids Back to RNA

By Patrick Ekeren, Aidan Klein, Hannah Mallari, Jennifer Pfaff

Codon degeneracy is great for handling mutations in our code, but can cause problems for informaticists. There are multiple ways to create a proteins from RNA codons, and our program takes in a sequence from the user and generates probable sequences of RNA that is made up of the amino acid inputs. Each single letter input is processed through an amino acid hashmap. The user is provided with a text file output with the sequences for the informacist to review.

Input:

The user inputs the sequence with a simple string of one letter protein codes. Our `get_seq` method automatically converts all characters to uppercase. If the input is entered blank, or with invalid entries, it will call itself to ask for a new input from the user.

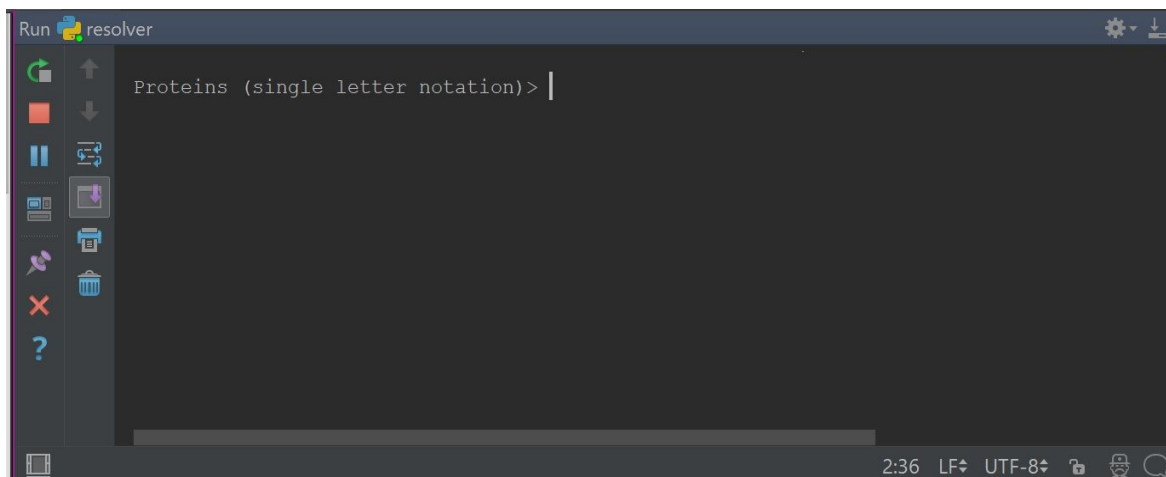


Fig 1. The user is prompted to input their sequence

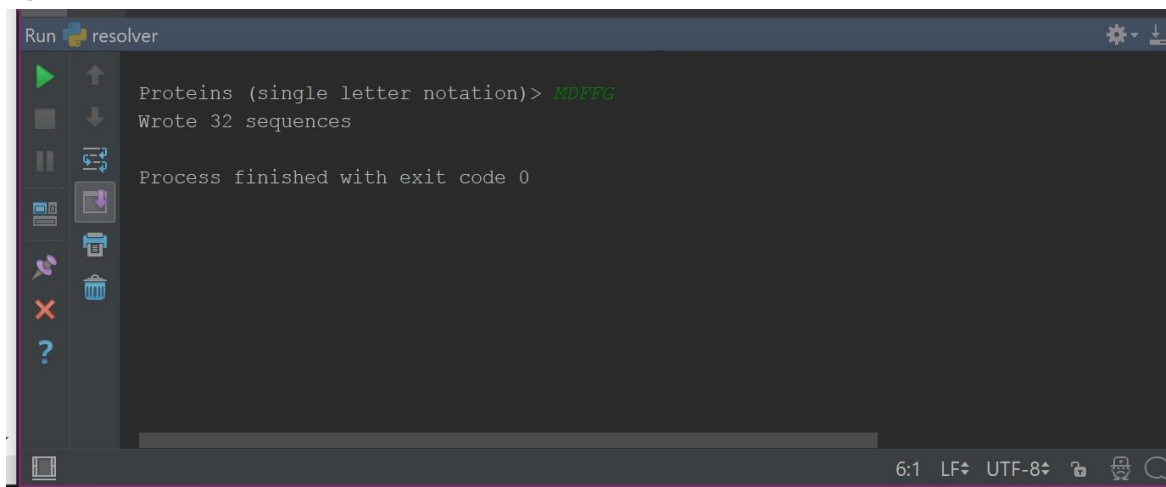
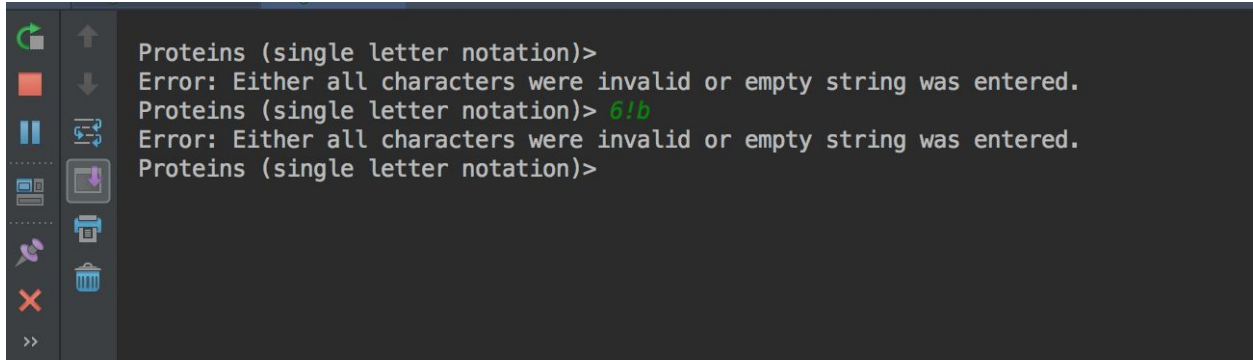


Fig 2. The program informs the user of the number of sequences outputted to the file.

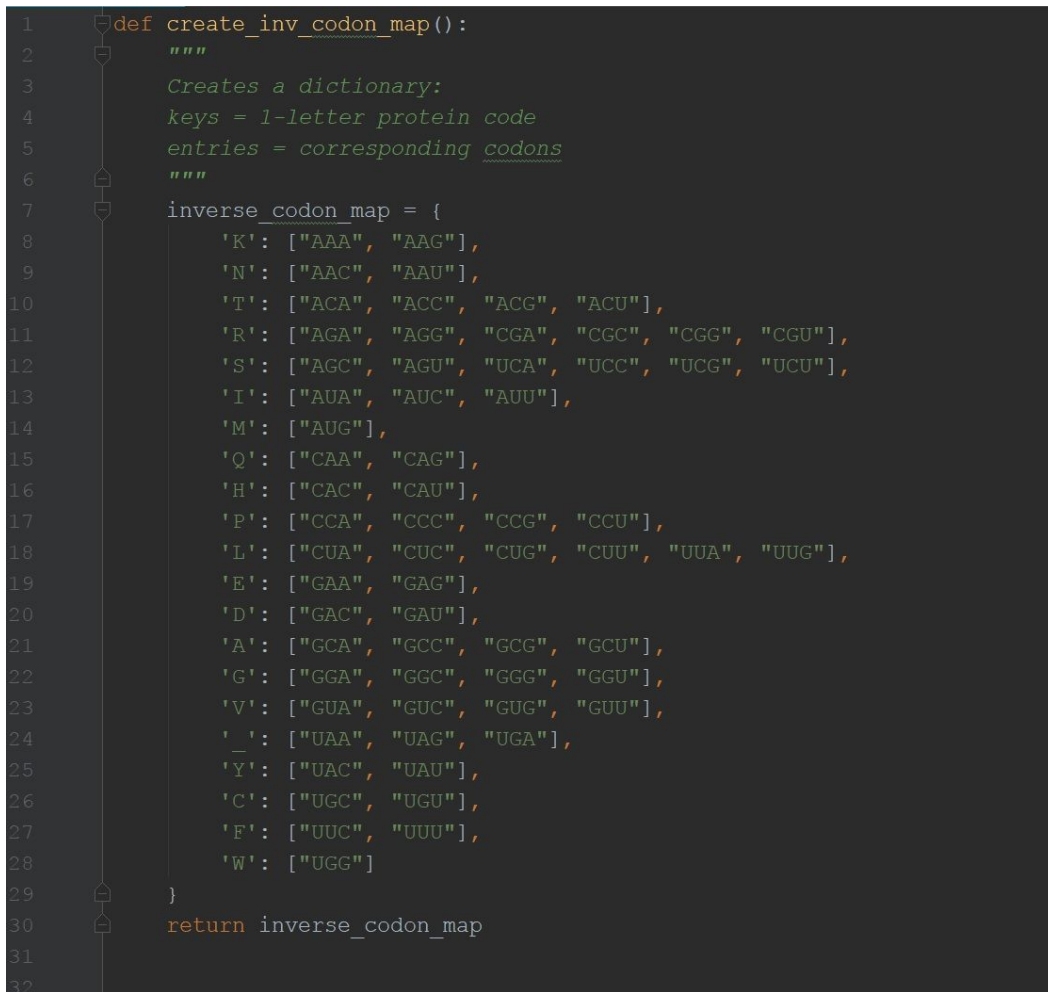


```
Proteins (single letter notation)>
Error: Either all characters were invalid or empty string was entered.
Proteins (single letter notation)> 6!b
Error: Either all characters were invalid or empty string was entered.
Proteins (single letter notation)>
```

Fig 3. If invalid inputs are entered, the program asks the user for a new input.

Invalid characters include empty spaces or anything that is not a protein 1-letter code. It is assumed the user knows the codes or is trying to parse a string of codes from some other source.

Codon Lookup Table:



```
1 def create_inv_codon_map():
2     """
3     Creates a dictionary:
4     keys = 1-letter protein code
5     entries = corresponding codons
6     """
7     inverse_codon_map = {
8         'K': ["AAA", "AAG"],
9         'N': ["AAC", "AAU"],
10        'T': ["ACA", "ACC", "ACG", "ACU"],
11        'R': ["AGA", "AGG", "CGA", "CGC", "CGG", "CGU"],
12        'S': ["AGC", "AGU", "UCA", "UCC", "UCG", "UCU"],
13        'I': ["AUA", "AUC", "AUU"],
14        'M': ["AUG"],
15        'Q': ["CAA", "CAG"],
16        'H': ["CAC", "CAU"],
17        'P': ["CCA", "CCC", "CCG", "CCU"],
18        'L': ["CUA", "CUC", "CUG", "CUU", "UUA", "UUG"],
19        'E': ["GAA", "GAG"],
20        'D': ["GAC", "GAU"],
21        'A': ["GCA", "GCC", "GCG", "GCU"],
22        'G': ["GGA", "GGC", "GGG", "GGU"],
23        'V': ["GUA", "GUC", "GUG", "GUU"],
24        '_': ["UAA", "UAG", "UGA"],
25        'Y': ["UAC", "UAU"],
26        'C': ["UGC", "UGU"],
27        'F': ["UUC", "UUU"],
28        'W': ["UGG"]
29    }
30    return inverse_codon_map
31
32
```

Fig 4. Hashmap for the codon possibilities

Codons possibilities are looked up from a table. This table maps proteins to their codons with all synonymous codons stored as a list. The program will use these lists to generate all sequences. We chose a dictionary to store this information because it is the quickest and easiest way to find corresponding “meanings” when given a value. (ex. The “M” protein means the codon “AUG” was used in the RNA).

Sequence Generation:

```
37
38 def get_seq(inverse_codon_map):
39     """
40     Asks user for input of a string with 1-letter protein codes, saved in variable proteins.
41     Goes through protein string and makes a list with just the valid protein codes.
42     If list is empty, either an empty string or invalid characters were entered, so it calls itself to ask for new input
43     :param inverse_codon_map: dictionary with inverse codons
44     :return: list of valid protein 1-letter codes
45     Contrib.: Aidan K., Hannah M.
46     """
47     proteins = input("Proteins (single letter notation)> ")
48     proteins = [p for p in list(proteins.upper()) if p in inverse_codon_map.keys()]
49     if len(proteins) == 0:
50         print("Error: Either all characters were invalid or empty string was entered.")
51         return get_seq(inverse_codon_map)
52     else:
53         return proteins
54
```

Fig 5. Input check

The function `get_seq()` starts by asking the user for an input of a string with 1-letter protein codes. It goes through the string and checks if each character is a key in the `inverse_codon_map` dictionary, and if it is, adds it to a new list. If the list of valid codes is 0, either an empty string or a string with only invalid characters was entered, so the function calls itself and asks for a new input. Returns the list of valid codes.

```
49
50 def generate_sequences(sequence: list, inverse_codon_map) -> list:
51     """
52     Generates a list of DNA sequences from a given protein sequence
53     Contrib.: Aidan K.
54     """
55     sub_sequences = ['']
56     if len(sequence) > 1:
57         sub_sequences = generate_sequences(sequence[1:],
58                                           inverse_codon_map) # Breaks the sequence down recursively into subsequences
59     gen_sequences = []
60     for codon in inverse_codon_map[sequence[0]]: # Builds each sequence into an array of output sequences
61         for sub_sequence in sub_sequences:
62             gen_sequences.append(codon + sub_sequence)
63     return gen_sequences
64
```

Fig 6. Sequence generator method

The function `generate_sequences()` takes in the list of valid codes (`sequence`) and the inverse codon map, and recursively goes through each character in `sequence` and adds them to a list, `sub_sequences`. Goes through each character in `sequence` again, uses each character as a key

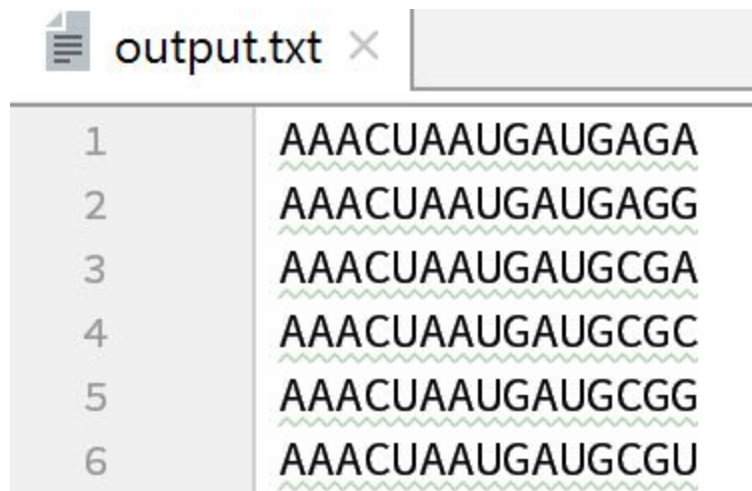
in the dictionary to get to the entries, and adds the entry to a new list, `gen_sequences`. For each entry, a new list is made. All possible codon combinations are returned.

Output Code and Format:

```
65 def output(outputsequence):
66     """
67     Outputs the sequences to a text file
68     Mostly written by Patrick with help cleaning up the code by Aidan
69     """
70     from os import linesep
71     text_file = open("output.txt", mode='w') # Creates text file to store the RNA sequence
72     string_output = linesep.join(outputsequence) # turns the array input into a string that can be written to the text file
73     text_file.write(string_output) # Writes the output to the output text file
74     text_file.close()
```

Fig 7. Output function

The output is placed into an output file named `output.txt` that is in the same directory as the script. Every new possible sequence is a new line in the text file. The `linesep` function is used to convert the data stored in an array into a string output that can be written into a file.



1	<u>AAACUAAUGAUGAGA</u>
2	<u>AAACUAAUGAUGAGG</u>
3	<u>AAACUAAUGAUGCGA</u>
4	<u>AAACUAAUGAUGCGC</u>
5	<u>AAACUAAUGAUGCGG</u>
6	<u>AAACUAAUGAUGCGU</u>

Fig 8. The text file output, `output.txt`, lists all possible sequences in an easy to read list