Project documentation and specification

1. Introduction

The goal of this document is to give an in-depth explanation of the design choices and the overall structure of the WebApp and the code.

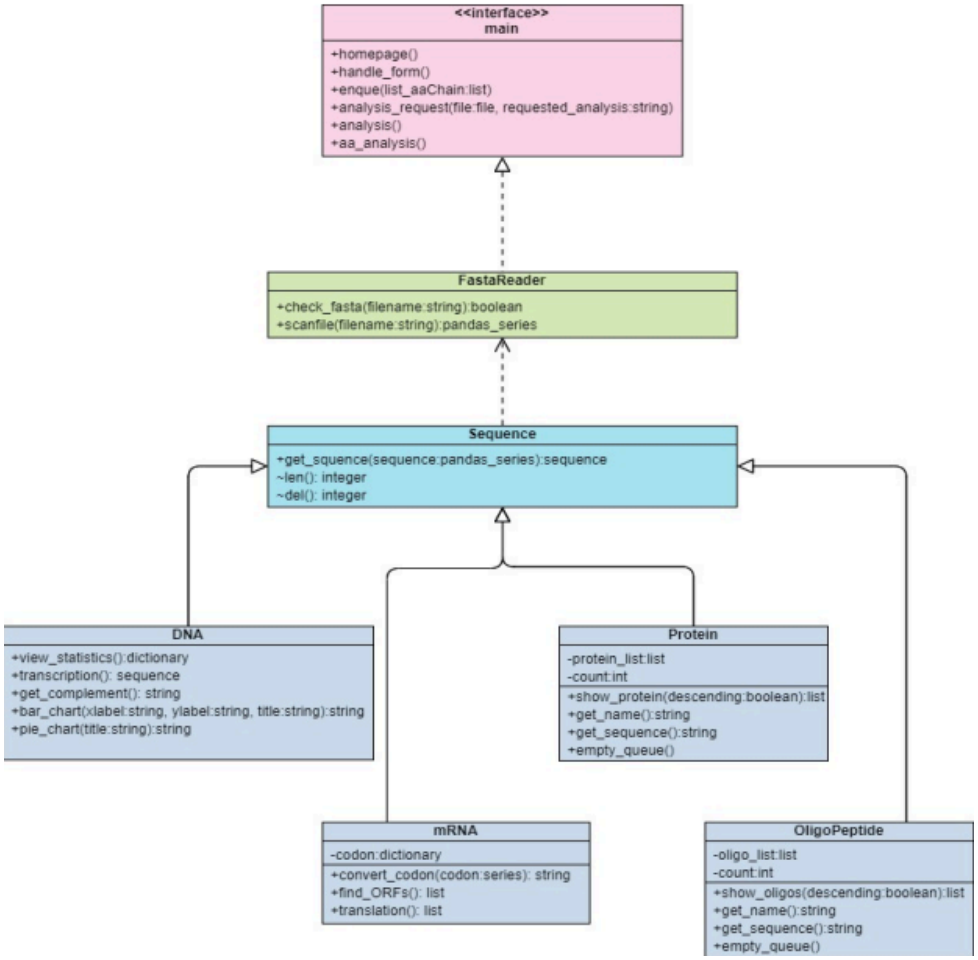
The stated goal of the project is to create an app that can read a FASTA document, show its content, transcribe it and translate it into amino acid sequences. In order to do so the present program takes as input a genome sequence in FASTA format and then proceeds to generate a Web-based UI where the genetic code is displayed, analyzed, transcribed and translated.

The code is organized in three main parts: the Fasta reader, the operators on the sequence and the flask script. The flask script is the linker between the HTML scripts that create the UI in WebApp.

The program follows the central dogma of molecular biology, beginning with the DNA sequence, followed by transcription to create mRNA, which is then translated into an amino acid chain according to the codon code.

Unfortunately a broader generalization is hard with little time and resources because nature is wide and diverse, but this program tries to reach as closely as possible the actual reality for viruses (the content of the Fasta file provided as input).

2. Organization



In the UML diagram the *Main* is at the top of the chain and it contains the flask script, because the rest of the classes are dependent on it.

Only when a file is provided in input, the class *FastaReader* is called in order to check the extension and the content of the file. If there is a positive response by the reader, the *Sequence* superclass will start operating on the pandas series obtained by the parses.

Sequence is a container for all the subclasses that help to process the data obtained by the file. The subclasses are four and each of them operates on a precise biological aspect.

The first subclass that elaborates the series is *DNA* and it transcribes the DNA and shows its statistics, then there is *mRNA* and it is crucial for the translation process and finally there are two twin classes that are involved in the amino acid chains managing and displaying: *Protein* and *OligoPeptide*. Although the subclasses can be considered dependent on one another, they only rely on *Sequence* to work together.

In view of the above, a pyramid like structure emerged and this was the only choice it was deemed perfect for the project due to its efficiency for the data stream from the file to the final output.

Non-python scripts and their interactions between the interfaces are not represented in the diagram because. These files are part of the HTML build connected to the Main script and they are divided for their role in the UI: homepage (homepage.html), DNA analysis and displaying (analysis.html) and transcript and translation displaying (aa_analysis.html). This subdivision was chosen in order to better organize the scripts and for user friendliness on app level. In order to smoothen out the HTML only UI, we created a CSS script (style.css) that implemented better features and looks to the webpages.

3. The classes

3.1 FastaReader

The file taken into input is processed by *FastaReader* and this class relies on pandas and os libraries in order to process the file.

The method that checks the file is called *scanfile* and, thanks to the os features, the file path is retrieved and passed as a variable to the *check_fasta* method; then the method will make sure that the file extension, the header and the content follow the correct format.

These checks are needed in order to be sure the file is in FASTA format both in extension and in format: all FASTA files begin with a header line starting with “>” containing the label of the sequence and optional information such as the species of origin. Meanwhile, the check on the content is necessary because the program works only with DNA information and therefore any different base or non canonical base can create unwanted errors later down the line.

If the check is passed, *scanfile* will continue and start reading the file and storing the genome inside a pandas series format that will be returned.

In addition both methods are static so they do not modify the object, so they simply process the data given as input and give the pandas series as output.

Class name: FastaReader	**Superclass:**	**Subclasses:**
Responsibilities	**Collaborations**	
check_fasta	main	
scanfile	check_fasta	

3.2 Sequence

The *Sequence* superclass is the container for the *DNA, mRNA, Peptide and OligoPeptide* subclasses.

The superclass is initialized with a pandas series and if the public method *get_sequence()* is used it will return what the object has stored.

Near this method there are two functions that are needed for returning the sequence length and resetting variables, attributes or methods inside the object itself if needed; these two functions are respectively *__len__()* and *__del__*().

Class name: Sequence	**Superclass:**	**Subclasses:** DNA, mRNA, Protein, Oligopeptide
Responsibilities	**Collaborations**	
get_sequence	main	
len		
del		

3.3 DNA

The *DNA* subclass is responsible for analyzing a sequence object and through inheritance the initiation is performed.

This class can fulfill various tasks, the most important of which is transcribing the sequence and in order to do this it is assumed that the plus strand was provided as input and then the Ts are substituted with Us in the *transcription()* method.

All the other tasks revolve around getting the statistics from the sample and this is done by the methods *view_statistics(), bar_chart(xlabel, ylabel, title)* and *pie_chart(title)*. The first among the three counts each type of nucleotide and draws frequencies and the most and least frequent and at last it returns a dictionary with all the information found. *bar_chart* and *pie_chart* are the methods responsible for drawing respectively a bar chart and a pie chart with the graphical representation of the frequencies of each nucleotide type through matplotlib library functions.

Unfortunately the integration between matplotlib and HTML is a bit rough and creates particular warnings that can spiral into crashes in some operating systems, like macOS. The solution around this problem was to implement base64 and io libraries in order to transform the plots into string that will be read as images into the web page, furthermore the use of the “matplotlib.use(‘Agg’)” in the code ironed out any left over warning.

Lastly there is a method that was not implemented in the *Main* and it is the *get_complement()*: this function returns a string with the complementary strand to the one given as input through list comprehension.

Class name: DNA	**Superclass:** Sequence	**Subclasses:**
Responsibilities	**Collaborations**	
transcription	Sequence	
get_complement		
bar_chart		
pie_chart		
view_statistics		

3.4 mRNA

The mRNA class is the second Sequence subclass, therefore we can initialize it by passing the sequence object obtained via transcription.

In this class the codons for the translation are stored as attributes in a dictionary with the codons as keys and the single letter abbreviation for the amino acid as values.

After initialization it is possible to use the method *find_ORFs()* in order to start the division in codons. The function converts the sequence object (a pandas series) into a list to better handle the search and when a "AUG" site is found every following triplet is turned into a string through the static function *convert_codon()* and added to a list. When a stop codon is detected then the list is appended into a list containing all the other meaningful sequences.

Then *translation()* parses all the open reading frames found by the previous operation into amino acid chains, using the dictionary attribute.

Class name: mRNA	**Superclass:** Sequence	**Subclasses:**
Responsibilities	**Collaborations**	
convert_codon		
find_ORFs	transcription, convert_codon, codon	
translation	find_ORFs	

3.5 Protein and OligoPeptide

These two subclasses can be considered twins because they do have the same exact methods, functions and attributes but they only differ in name.

Both classes have a list and a counter as attributes, these attributes are initialized at zero and only when the Protein or OligoPeptide object is initialized the list is updated with the values given as input and the counter starts going upwards. In the initiation process the counter is unified into a string with the word “Protein” or “OligoPeptide” in order to have a more pleasing display to the user.

Then there are two methods that return two different strings: one is a string of the sequence object (*get_sequence()*) meanwhile the other one returns the name of the protein/oligopeptide (*get_name()*).

After this every other method is static by design choice, so that no modification was done by the methods to the attributes because it is critical for them to remain unchanged and not alter the name-chain association.

The main protagonist of these two classes is the *show_proteins/oligos()* method and it is responsible for showing either in ascending or descending order through a boolean flag given in input. Meanwhile on the other side there is the *empty_queue()* operation that once it is run it resets the object to zero through the *__del__()* function inherited from *Sequence*.
Besides these two crucial methods there is a minor function that serves to simply return the list: *get_list_protein/oligo()*.

Class name: Protein	**Superclass:** Sequence	**Subclasses:**
Responsibilities	**Collaborations**	
show_proteins	list_proteins, count, main	
get_name		
get_sequence		
empty_queue	list_proteins, count	
get_list_protein		

Class name: OligoPeptide	**Superclass:** Sequence	**Subclasses:**
Responsibilities	**Collaborations**	
show_oligos	list_oligos, count, main	
get_name		
get_sequence		
empty_queue	list_oligos, count	
get_list_oligo		

4. Main description

The *Main* script imports all the previously mentioned classes and creates the user interface at runtime by using the Flask framework. In the homepage the program will ask the user to upload a FASTA file and select whether to analyze the sequence or view the amino acid chains. The file and action inputs are taken in by the *handle_form* function using the POST method and it will redirect the user to the appropriate page depending on the chosen action.

The *analysis_request* function is called in the routing functions for both analysis.html and aa_analysis.html. It creates a DNA object with sequence obtained from FastaReader and runs all of the operations necessary to obtain the information the user requested. It also takes care of error handling, by making sure the file has the correct format and is within the "fasta_file" directory.

Two additional functions manage the amino acid chains: enqueue, which sorts them into oligopeptides and proteins based on their length, and empty_queue which resets the internal lists contained in both the OligoPeptide and Protein classes to allow for another file to be examined.

5. Simplification and limitations

The translation process is quite difficult to generalize, this is due to the complexity of reading frames in nature. In general on a double stranded DNA there are 6 reading frames and only one is then transcribed and translated, unfortunately the management of those varies between prokaryotes and eukaryotes and a wide and broad generalization with little time and resources is impossible.

Unfortunately viruses are not living beings and they can "violate" the rule: in fact the need for viruses to store the proteins into a small stretch of genetic information made them capable of creating overlapping reading frames with frameshifts[1].

This mechanism of frameshifts and overlapping ORFs is present for example is present in the two genomes present in fasta_file directory: Sars-Cov-2[2] (the sample data provided) has a notable -1 frameshift in the middle and some overlapping ORFs and in HIV[3] (the other named sample) has multiple overlapping frames.

In light of the above, it was necessary to do simplifications during the translation process and use only one reading frame and do not use overlapping ORFs. Although the simplifications may have impacted the fidelity of the program, it was able to find 23 chains against the 29[4] known so far in Sars-Cov-2 and 18 out of 20[5] in HIV.

6. Conclusions

This project may not be the accurate portrayal of the complex reality genetic variety both in living beings and in viruses, but tries at its best to recreate as close as possible the actual truth.

The app could be considered a good starting point for the future and with more time, resources and studying on viral genetics, the simplification could be solved and the program may turn into a useful instrument for viral genomic analysis.

---

[1] Chapter 1.1, Analysis of overlapping reading frames in viruses, Dr. R. Hilgenfeld, 2015
[2] The coding capacity of Sars-Cov-2, Nature, 9/09/2020
[3] The biased nucleotide composition of the HIV genome: a constant factor in a highly variable virus, Retrovirology, Biomedical Central, 6/11/2012
[4] Overview of SARS-CoV-2 genome-encoded proteins, National Library of Medicine, 10/08/2021
[5] Human Immunodeficiency Virus (HIV), National Library of Medicine, 9/05/2016