UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE USTHB
DEPARTEMENT D'INFORMATIQUE

Project TP SPS

A Python Code Documentation for DNA Sequence Analysis

Présenté par :
GUELFEN Abdelheq

Année universitaire 2023-2024

Contents

1 Introduction

This is a python code documentation about a program that automate the proccess of analysing sets of DNA.
Source code is seperated into 3 directories and 2 files:

- **data**: data is the folder where our DNA sequences are stored
- **modules**: This folder has three objects, each with a specific role
- **output**: output is the folder where our results are stored
- **main.py**: This is our main entry point to the program
- **variables.py**: This file is where global variables are stored and will be useed throught the lifetime of the program

2 Main file (main.py)

We are importing all the modules we need from dir:modules

```
from modules.dna_generator import DNA_Genrator
from modules.dna_processor import DNA_Processor
from modules.protein_builder import Protein_Builder

from variables import results
```

The function starts by showing an interactive menu with multiple choices to choose from.
User should choose a number from (1-11) according to the menu.

```
def main():
    print(
        """
 1: Generate a DNA
 2: Validate the DNA
 3: Calculate the necleotides base frequency
 4: Translate DNA to RNA
 5: Translate RNA into a protein
 6: Calculate the inverse complement of a DNA
 7: Calculate the taux of GC in a DNA
 8: Calculate the frequency of a codons in a DNA
 9: Make a random mutations to DNA
10: Find a motif in a DNA
11: Save results in a file
"""
    )
```

An infinite loop is used to keep the program running, for every number chosen a specific function gets executed to solve that problem

```
while True:
    choice = input("> ")

    if choice == "quit()":
        break

    if choice == "1":
        DNA_GENRATOR().generate()

    if not results.get("DNA"):
        print("[ERROR]: Please generate a set of DNA first")
        continue

```

```python
    if choice == "2":
        DNA().validate()
        print(f"DNA is validated: {results['is_dna_validated']}", "\n")

    if choice == "3":
        ...
```

The program can not manipulate DNA without generating a sequence first, so this condition limit and warn user to generate a sequence of DNA first

```python
if not results.get("DNA"):
    print("[ERROR]: Please generate a set of DNA first")
    continue
```

3 DNA generator(modules/dna_generator.py)

3.0.1 Function generate()

There are two type of generating a DNA sequence, either randomly or from a file.
The function prints a submenu and take a choice from the user and execute one of the two functions **generateRandomly()** or **chooseFromFile()**.
The input is limited to **(a, b or 0)** by using the *assert* keyword.
The user can choose 0 to go back to main menu

```python
def generate(self):
    print(
        """a: generate a random set
b: choose from a predefined sets
0: go back to main menu
    """
    )

    while True:
        choice = input("(1: generate DNA)>> ")

        try:
            assert choice in ("a", "b", "0")

            if choice == "a":
                results["DNA"] = self.generate_randomly()
                print(results["DNA"], "\n")
                break

            if choice == "b":
                results["DNA"] = self.choose_from_file()
                print(results["DNA"], "\n")
                break

            if choice == "0":
                break

        except (AssertionError, ValueError):
            print("ERROR: Please enter a valid choice")
```

3.0.2 Function generateRandomly()

This function takes an input(number of sequences to generate) and return a DNA sequence.
We choose a Random necleotide and save it in a list for the number of choice entered by user, this has been implemented using list comprehension (simple way of writing for loop).
".join() is used to trnasform a list to a string.

```python
def generate_randomly(self):
    while True:
        print("how many sequence do you want?")
        choice = input("(1: generate DNA)>> ")

        try:
            assert choice.isdecimal

            return "".join([random.choice(SYMBOLS) for _ in range(int(choice))])
        except (AssertionError, ValueError):
            print("ERROR: Please enter a valid number\n")
```

3.0.3 Function chooseFromFile()

This function presents a menu of DNA sequences, reads them from a file, and allows the user to choose one by entering a corresponding number. It validates the input and returns the selected DNA sequence. If the input is invalid, it prints an error message and prompts the user to enter a valid number.

```python
def choose_from_file(self):
    print(
        """1: sequence 1 consisting of 1000 bases.
2: sequence 2 consisting of 1000 bases.
3: sequence 3 consisting of 500 bases.
4: sequence 4 consisting of 100 bases.
5: sequence 5 consisting of 700 bases.
    """
    )

    sequences = self.read_fasta()

    while True:
        choice = input("(1: generate DNA)>> ")
        try:
            assert choice.isdecimal

            return sequences[int(choice) - 1]

        except (AssertionError, ValueError):
            print("ERROR: Please enter a valid number\n")
```

4 DNA processor(modules/dna_processor.py)

4.0.1 Method validate()

The **validate** method checks if the DNA sequence stored in the DNA key of the results dictionary consists only of valid symbols specified in the SYMBOLS set. It sets the **isDnaValidated** key in the results dictionary to indicate whether the DNA sequence is valid or not.

```python
def validate(self):
    is_validated = set(results.get("DNA", "")).issubset(SYMBOLS)

    results["is_dna_validated"] = is_validated
```

4.0.2 Method calculateFrequency()

The **calculateFrequency** method prompts the user to enter a nucleotide base (A, T, C, or G) to calculate its frequency in a DNA sequence. The user can enter '0' to return to the main menu. The method validates the input, ensuring it is a valid nucleotide base, and then calculates and prints the frequency of the entered base in the DNA sequence by using **count()** method. If an invalid input is provided, it catches the assertion error or value error, prints an error message, and prompts the user to enter a valid nucleotide base.

```python
def calculate_frequency(self):
    print("Enter necleotide to calculate frequency (A,T,C,G)")
    print("Enter 0 to go back to main menu\n")

    while True:
        try:
            base = input("(3: Calculate frequency)>> ")
            if base == "0":
                print()
                break

            assert base.upper() in SYMBOLS

            print(f"frequency of {base} is : {results.get('DNA','').count(base)}\n
    ")

        except (AssertionError, ValueError):
            print("[ERROR]: Enter a valid necleotide base\n")
```

4.0.3 Method translateToRNA()

The **translateToRNA** method takes the DNA sequence stored in the "DNA" key of the results dictionary, replaces all occurrences of "T" with "U" using **replace()** method and then stores the resulting RNA sequence in the "RNA" key of the results dictionary.

```python
def translate_to_RNA(self):
    results["RNA"] = results.get("DNA", "").replace("T", "U")
```

4.0.4 Method calculateInverse()

The **calculateInverse** method generates the DNA inverse sequence by mapping each nucleotide in the original DNA sequence to its complementary base using the **COMPLEMENT** dictionary. It then joins these complementary bases together to form the inverse sequence. The resulting DNA inverse sequence is stored in the **DNAInverse** key of the results dictionary. **map** function in Python applies a given function to each item of an iterable, it does the same thing as **for loop**

```python
def calculate_inverse(self):
    inverse = "".join(
        map(lambda base: COMPLEMENT.get(base, ""), results.get("DNA", ""))
    )

    results["DNA_inverse"] = inverse
```

4.0.5 Method calculateGCContent()

The **calculateGCContent** method computes the GC (Guanine-Cytosine) content percentage of a DNA sequence. It retrieves the DNA sequence from the results dictionary, counts the occurrences of "G" and "C,", then stores the result as a formatted string in the **GCcontent** key of the results dictionary.

```python
def calculate_GC_content(self):
    dna = results.get("DNA", "")

    gc_content = (dna.count("G") + dna.count("C")) / len(dna) * 100
    results["GC_content"] = f"{gc_content}%"
```

4.0.6 Method makeMutations()

The **makeMutations** method prompts the user to enter the number of random mutations they want to introduce into a DNA sequence. It then performs the specified number of mutations by randomly selecting nucleotide positions and replacing the corresponding bases with other randomly chosen nucleotides. The method ensures that each mutation results in a different nucleotide from the original sequence. The updated DNA sequence is stored in the "DNA" key of the results dictionary.

```python
def make_mutations(self):
    print("Enter the number of mutation you want to make:")

    dna = results.get("DNA", "")

    while True:
        try:
            choice = input("(9: Random mutations)>> ")
            assert choice.isdecimal

            random_indexes = []
            for _ in range(int(choice)):
                random_necleotide = random.choice(SYMBOLS)
                random_index = random.randint(0, len(dna) - 1)

                while random_index in random_indexes:
                    random_index = random.randint(0, len(dna) - 1)
                random_indexes.append(random_index)

                while dna[random_index] == random_necleotide:
                    random_necleotide = random.choice(SYMBOLS)

                dna = list(dna)
                dna[random_index] = random_necleotide
                results["DNA"] = "".join(dna)

            print(f"new mutated DNA:\n{results.get('DNA','')}")
            break

        except (AssertionError, ValueError):
            print("[ERROR]: Enter a valid necleotide base\n")
```

4.0.7 Method findMotif()

The **findMotif** method prompts the user to enter a DNA motif they want to search for within a given DNA sequence. It handles the input, ensuring the motif consists of valid nucleotide bases. The method then uses regular expressions (re.finditer) to find all occurrences of the motif in the DNA sequence and prints the indexes where the motif is found. The user can enter '0' to return to the main menu.

```python
def find_motif(self):
    print("Enter motif you want to search for")
    print("Enter 0 to go back to main menu\n")

    while True:
        try:
            motif = input("(10: Find motif)>> ")
            if motif == "0":
                print()
                break

            for i in list(motif):
                assert i.upper() in SYMBOLS

            occ = [
                m.start()
                for m in re.finditer(f"(?={motif})", results.get("DNA", ""))
            ]
            print(f"This motif has found at these indexes: {occ}\n")

        except (AssertionError, ValueError):
            print("[ERROR]: Enter a valid necleotide base\n")
```

5 Protein builder(modules/protein_builder.py)

5.0.1 Method generateFromRna()

The **generateFromRna** method checks if an RNA sequence is present in the results dictionary. If not, it translates the DNA sequence to RNA. It then searches for protein-coding sequences in the RNA using a specified pattern and translates them into amino acid sequences using a codon table. The resulting protein sequences, starting with the "M" (methionine) codon, are stored in the results dictionary under the "proteins" key.

```python
def generate_from_rna(self):
    if not results.get("RNA"):
        DNA().translate_to_RNA()

    pattern = r"AUG((?:.{3})*?)(UAA|UAG|UGA)"
    matches = re.findall(pattern, results.get("RNA", ""))

    protiens = []
    for seq, _ in matches:
        sequences = [seq[i : i + 3] for i in range(0, len(seq), 3)]
        acide_amino = map(lambda seq: codon_table.get(seq, ""), sequences)

        protiens.append("M" + "".join(acide_amino))

    results["proteins"] = protiens
```

5.0.2 Method calculateCodonsFrequencey()

The **calculateCodonsFrequencey** method checks if protein sequences exist in the results. If not, it generates them using the **generateFromRna** method. Then, it calculates and records the frequency of each unique amino acid in the proteins. The results are stored in the results dictionary under the key "codonsFrequency".

```python
def calculate_codons_frequencey(self):
    if not results.get("proteins"):
        self.generate_from_rna()

    codons_frequencey = {}

    for protein in results.get("proteins", []):
        for acido_amino in set(protein):
            if codons_frequencey.get(acido_amino) is None:
                codons_frequencey.setdefault(
                    acido_amino, protein.count(acido_amino)
                )
            else:
                codons_frequencey[acido_amino] = protein.count(
                    acido_amino
                ) + codons_frequencey.get(acido_amino, 0)

    results["codons_frequency"] = codons_frequencey
```

6 Variables file (variables.py)

This file contains global variables that will be used throught the execution of the program.

```python
results = {
    "DNA": "",
    "DNA_inverse": "",
    "RNA": "",
    "proteins": [],
    "is_dna_validated": False,
    "GC_content": "",
    "codons_frequency": "",
}

# Global constants
SYMBOLS = ("A", "T", "C", "G")
COMPLEMENT = {
    "A": "T",
    "T": "A",
    "G": "C",
    "C": "G",
}
codon_table = {
    "UUU": "F",
    "UUC": "F",
    "UUA": "L",
    "UUG": "L",
    "CUU": "L",
    ...
}
```