Homework 2: DNA Sequence Manipulation

Implement a Python program that performs various operations on DNA sequences using classes and inheritance. You will be working with the concepts of object-oriented programming to create classes, define methods, and utilize inheritance.

Requirements:

Part I: Create file `DNA_sequence.py`. In this file, create the following classes:

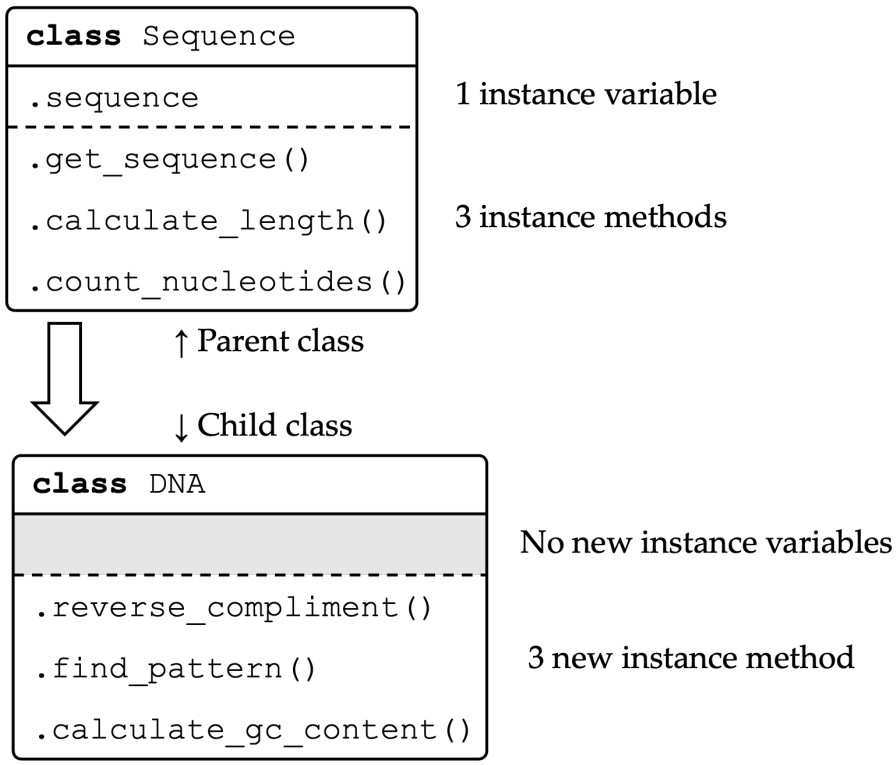

Figure 1: A simplified view of the inheritance structure for this assignment. Note that we omit the `init` dunder method from this diagram, but you will need it to attach instance variables to new objects."

Class 1: Create a base class called `Sequence` with the following attributes:

Instance variables:

- `sequence` (string): Stores the DNA sequence.

Methods:

- `get_sequence(self)`: Returns the DNA sequence.
- `calculate_length(self)`: Calculate and return the length of the sequence
- `count_nucleotides(self)`: Count and return a dictionary that contains the number of each nucleotide (A, T, C, G) in the sequence.

Class 2: Create a subclass called DNA that inherits from the `Sequence` class. Implement the following methods in the DNA class:

- `reverse_complement(self)`: Returns the reverse complement of the DNA sequence. The reverse complement is obtained by reversing the sequence and replacing each nucleotide with its complement (A with T, T with A, C with G, and G with C).
- `find_pattern(self,pattern)`: Returns the starting indices of all occurrences of a given pattern in the DNA sequence.
- `calculate_gc_content(self)`: Calculates and returns the GC content of the DNA sequence as a percentage.

  Note: The goal of the `calculate_gc_content` method is to determine the GC content of the DNA sequence as a whole, not as individual counts of "G" and "C." This means that "GC" should be treated as a single entity when calculating the percentage. Suppose we have the DNA sequence `"ATATGCGCGTGC"`, the GC content is (6 / 12) * 100 = 50.0%.

Part II: Test using `if __name__ == '__main__'`

Create an instance of the "DNA" class with a specific DNA sequence and use it to demonstrate your methods with print statements in an `if __name__ == '__main__':` block:

```
if __name__ == "__main__":
    # Testing
    dna_sequence = "ATGCAAGG"

    # Create an instance of the DNA class
    dna = DNA(dna_sequence)

    # Test the implemented methods
    seq_length = dna.calculate_length()
    count_nucleotid = dna.count_nucleotides()
    reverse_complement = dna.reverse_complement()
    pattern_indices = dna.find_pattern("GG")
    gc_content = dna.calculate_gc_content()

    print("Original sequence:", dna.get_sequence())
    print("sequence length: ", seq_length)
    print("sequence nucleotides: ", count_nucleotid)
    print("Reverse complement:", reverse_complement)
    print("Pattern indices:", pattern_indices)
    print("GC content:", gc_content)
```

- Demonstrate the usage of all 6 implemented methods, such as retrieving the sequence, calculating the length, counting nucleotides, transcribing the sequence, and obtaining the reverse complement.
- Use comments to denote new tests
- Do not reuse the sequence provided in the example above

If done correctly, your `DNA_sequence.py` should produce something like the following when executed (though

of course your DNA sequence will be unique):

```
$ python3 ./DNA_sequence.py
Original sequence: ATGCAAGG
sequence length:  8
sequence nucleotides:  {'A': 3, 'T': 1, 'C': 1, 'G': 3}
Reverse complement: CCTTGCAT
Pattern indices: [6]
GC content: 25.0
```

Part III: Test using unittest

Print statements are nice for quickly debugging code, but unittests are ultimately better. Create a file `test_DNA_sequence.py` to test the functionality of **all** the previously created classes and their methods

Here is an example for test `get_sequence` for `Sequence.get_sequence()`:

```
import unittest
from DNA_sequence import Sequence, DNA

class TestSequence(unittest.TestCase):
    def setUp(self):
        """Attach attributes to TestCase object that are created fresh at the top
        of every test method below."""
        self.sequence = Sequence("ATGCA")
        self.sequence2 = Sequence("")

    def test_get_sequence(self):
        """test method get_sequence that returns the DNA sequence."""
        # Test a known sequence
        self.assertEqual(self.sequence.get_sequence(), "ATGCA")

        #test an empty sequence
        self.assertEqual(self.sequence2.get_sequence(), "")


unittest.main()
```

Structuring unittests is a bit of an art form - how many `unittest.TestCase` classes should you have? How big should a "unit" test be ("unit" means one "unit" of functionality, but that's a fuzzy definition in itself). For now, follow these guildines:

- Create 1 unittest.TestCase class for **each class** you are testing. This means your unittest file should have 1 class for testing `Sequence` and one for testing `DNA`.
- Include one test method for each method you are testing. This means your `Sequence` test class should have at least 3 methods - one each for testing `get_sequence`, `calculate_length`, and `count_nucleotides`.

  Similarly, your `DNA` test class should include three methods, one each for `reverse_complient`, `find_pattern`, and `count_gc_content`.

Readability

Every method should have a docstring. Follow best practices for naming conventions. Use comments to explain what your code does.

Imports

Do not import any modules besides `unittest` in this assignment. As always, you are free to import any modules you have written yourself (e.g. you can import `DNA_sequence` when testing, since that is a module you have written yourself).

Submitting

At minimum, submit the following files:

- `DNA_sequence.py`
- `test_DNA_sequence.py`

Students must submit to Gradescope individually by the due date (typically at 11:59 pm EST) to receive credit.

Grading

Part of your grade will be manual, and based on readability and style. Ensure that your code is well-structured, follows best practices, and includes appropriate docstring and comments.

Test your code with different DNA sequences to ensure its correctness and accuracy.

Broadly speaking, we grade coding assignments on 4 main areas: Structure, Tests, Efficiency, and Readability.