Programming II – Lab Activity: Four Coding Challenges

Instructions

In this lab, you will work through **four coding challenges**. Some challenges may include **optional bonus tasks**—complete them only after finishing the main tasks if time allows.

You may use **AI tools** (e.g., ChatGPT, Copilot) to assist you. However, you are responsible for understanding the final code and ensuring it follows all requirements below.

Rules & Requirements

1. **One Script per Challenge**

- Submit one **Python 3** script (.py file) for each challenge.
- Name files clearly (e.g., challenge1.py, challenge2.py, etc.).
- Start each script with a **shebang**:

```
#!/usr/bin/env python3
```

- Include a guard statement at the bottom:

```
if __name__ == "__main__":
    main()
```

2. Documentation & Type Hints

- All functions must include type hints and docstrings explaining inputs, outputs, and purpose.
- Add brief comments for any non-obvious logic.

3. Object-Oriented Design

- Each solution must use at least one class (see template below).

```
class ChallengeX:
    """
    A class to encapsulate the solution for Challenge X.
    """

    def __init__(self, data: Any):
        """
        Initialize the challenge with input data.
        :param data: Any input data needed for the challenge.
        """
        self.data = data
```

```
    def solve(self) -> Any:
        """
        Implement the main logic for the challenge.

        :return: The result of the computation.
        """
        # TODO: Implement the solution logic here
        return None
```

4. Unit Testing

- Include unit tests for your code in the same script.
- Use Python's unittest module (or pytest if specified), following the template below.

```
#!/usr/bin/env python3
"""
Challenge X - Programming II
Description: Briefly describe what this challenge does.
"""

from typing import Any
import unittest


class ChallengeX:
    """
    A class to encapsulate the solution for Challenge X.
    """

    def __init__(self, data: Any):
        """
        Initialize the challenge with input data.
        :param data: Any input data needed for the challenge.
        """
        self.data = data

    def solve(self) -> Any:
        """
        Implement the main logic for the challenge.

        :return: The result of the computation.
        """
        # TODO: Implement the solution logic here
        return None


def main() -> None:
    """
    Main function to run the challenge solution.
```

```
    """
    # Example usage
    challenge = ChallengeX(data="example input")
    result = challenge.solve()
    print(f"Challenge result: {result}")


class TestChallengeX(unittest.TestCase):
    """
    Unit tests for ChallengeX class.
    """

    def test_solve(self):
        """
        Test the solve() method with sample input.
        """
        challenge = ChallengeX(data="test input")
        self.assertEqual(challenge.solve(), None)  # Replace None with expected output


if __name__ == "__main__":
    main()
    unittest.main()
```

5. Bonus Challenges

- Attempt these only after completing the main challenges.
- Mark them clearly in your code comments (e.g., # BONUS SOLUTION).

6. Academic Integrity

- If you use AI assistance, you must still understand and be able to explain your solution.

Challenges

Challenge 1: Difficulty level 2/10: Malcolm in the middle

Write a function named `midaa` that takes a string of amino acids as its parameter. The string should be the single letter abbriviations which can be found easily online.

Your function should extract and return the middle amino acid. If there is no middle amino acid, your function should return the empty string.

For example, `midaa("MKTWQSL")` should return "W" and `midaa("AGCTYP")` should return "".

Challenge 2: Difficulty level 3/10: The one that counted fields

Define a function named `count` that takes a single parameter. The parameter is a string. The string will contain a fasta header line divided into fields by underscores, such as this:

">NODE_1_length_142_cov_30.24_cutoff_0"

For context, this is what the headerlines of the contigs output file look like when you use SPades.

Your function should count the number of fields (each field is separated by an underscore, do not forget) and return that number.

For example, the call count(">NODE_1_length_142_cov_30.24_cutoff_0") should return 8.

Bonus level 2.1: Other fields

Complete this only after completing the initial Challenge 2 above.

There are other ways headers are deliminated, for example when running MEGAHIT:

```
>contig_1 flag=multi_part len=12345 multi=3
```

Task: adapt your script to count fields based on " " instead of "_"

Bonus level 2.2: header strings

Complete this only after completing the initial Challenge 2 above.

Another important task that is seen often in bioinformatics is extracting information from header strings.

Task: Incorporate the creation of an output that shows what is in each field and its value. The output can be simple and print to screen or you can make a dictionary, your choice.

Bonus level 2.3: multiple entries

Complete this only after completing the initial Challenge 2 above.

There can be multiple entries in a fasta file, so lets address that.

Task: Write me a scrip that will combine all the above tasks and do it for each header in a fasta file (`20240912_input.fasta`; provided). The output should show information on all the headers in a neat compact way.

Challenge 3: Difficulty level 4/10: The data is not a bird but it is nested

Often times data can be multidimentional, like in the example table below.

Test 1: M | W | X

Test 2: M | Y | X

Test 3: "" | "" | ""

	A	**B**	**C**
Test 1	X	Y	X
Test 2		X	
Test 3	Y	Y	Y

Data like this can be made 2 dimentional, and represented like below (row and column names ignored for simplicity.

Note: there are many ways to nest data in python. There are nested dictionaries and tuples as well, which create a higher level of complexity.

tests = [["X", "Y", "X"], [" ", "X", " "], ["Y", "Y", "Y"],]

Imagine if as a user you enter "C1" and you need to see if there's an X or Y in that cell on the board. To do so, you need to translate from the string "C1" to row 0 and column 2 so that you can check tests[row][column].

Your task is to write a function that can translate from strings of length 2 to a tuple (row, column). Name your function get_row_col; it should take a single parameter which is a string of length 2 consisting of an uppercase letter and a digit.

For example, calling get_row_col("A3") should return the tuple (2, 0) because A3 corresponds to the row at index 2 and column at index 0 in the tests.

Bonus level 3.1: list comprehension

Do if you have completed everything in Challenge 3 above.

Task: Write another function (or build it into the one for the above task) to pull out and return the value that is in the position that was put in. For example, [0, 1, 2] becomes [0, 2] and back to [0, 1, 2].

Bonus lebel 3.2: multiple arguments

Do if you have completed everything in Challenge 3 above.

Task: Make a function called `add_value_to_test` that will add a value to a given position. So if I used `add_value_to_tests("tests", "A3", "X")` then it will add an X in the correct list and position.

Challenge 4: Difficulty level 5/10: Comparing sequences

Define a function named sequence_compare. Your function should take three parameters: gene, sequence1, and sequence2.

Your function must return whether gene is exclusively present in sequence1 or sequence2.

In other words, if gene is found in both sequences or in neither of the sequences, return False. If gene is found in only one of the sequences, return True.

```
sequence_compare('BRCA1', ['BRCA1', 'TP53', 'EGFR'], ['HER2', 'BRCA2', 'PTEN'])
sequence_compare('BRCA1', ['APOE', 'TP53', 'EGFR'], ['BRCA1', 'HER2', 'PTEN'])
sequence_compare('BRCA1', ['BRCA1', 'TP53', 'EGFR'], ['BRCA1', 'HER2', 'PTEN'])
sequence_compare('BRCA1', ['APOE', 'TP53', 'EGFR'], ['HER2', 'BRCA2', 'PTEN'])
```