QLC-1) - Find the Highest Number.

Objective

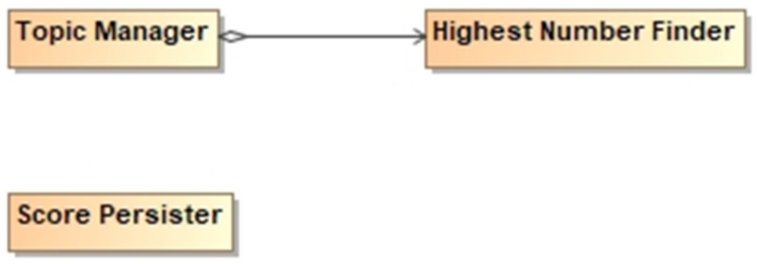
In this lab, you will incrementally develop a simple Python class called HighestNumberFinder using Test-Driven Development (TDD). You will write tests first, followed by the minimal production code needed to make those tests pass. Each step reflects the RED → GREEN → REFACTOR cycle of TDD. We will also follow a common pattern for structuring tests called Arrange-Act-Assert (AAA) that helps make tests more organised and readable.

Case Scenario: Storing Top Scores by Topic

An organisation delivers several topics (subjects). Students are graded on each topic. You are required to store the top score for each topic.

We've designed the application so that it comprises three core classes:

- A class to find the highest number from a list of integers.
- A class to find the highest score for a topic.
- A class to write the topic and score to a file on the disk.



Initial Lab Brief

In this initial lab, you will develop the tests and production code for the Highest Number Finder class.

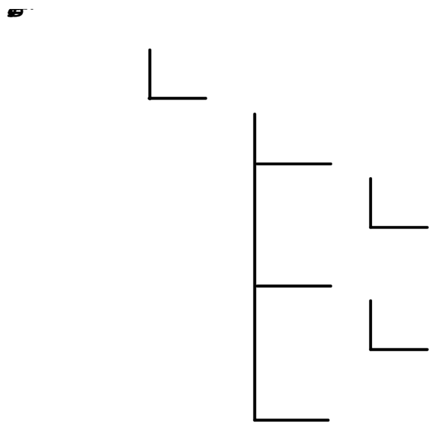
You are going to follow a **TDD** approach to finding the highest number in an array of integers

Given the following specification

- If the input were [4, 5, -8, 3, 11, -21, 6] the result should be 11
- An empty array should throw an exception (DO LAST)
- A single-item list should return the single item (DO FIRST)
  E.g. [10] should return 10.
- If several numbers are equal and highest, only one should be returned
- If the input were [7, 13] then the result should be 13
- If the input were [13, 4] then the result should be 13

Steps

1. **Create your Project Folder**
   Start by creating a new Python project named HighestNumberServices. This will also serve as your Git repository.
2. **Create the Core Structure**
   Inside HighestNumberServices, set up the following folder layout.



3. **Initialise the App and Test Packages**
   Ensure that both app and tests directories include an __init__.py file to make them recognizable as Python packages.
4. **Designate main.py as the launcher**
   Place main.py in the findhighestnumber folder (above app and tests). This script will serve as the launch point for running your application and as the root for test discovery.
5. **Create your first test file**
   Create a test script named test_highest_number_finder.py inside the tests folder.
   In this file:
   - Define a class called TestHighestNumberFinder (following PEP 8 naming conventions for test classes).
   - Ensure it inherits from unittest.TestCase, which provides the framework for writing unit tests in Python.
6. **Write your 1st test (RED)**
   Before handling complex logic or edge cases, begin with the simplest possible test case:
   - A single-item list should return that item as the highest number.
   - Your first test method should be named something like:
     def test_find_highest_in_list_of_one_expect_single__item(self):
   - Focus on getting this test to pass before moving on to more advanced scenarios.
   - Commit code to Git.

```
class TestHighestNumberFinder(unittest.TestCase):
  def test_find_highest_in_list_of_one_expect_single_item(self):
    # Arrange
    numbers = [10]
    cut = HighestNumberFinder()
    expected_result = 10

    # Act
    result = cut.find_highest_number(numbers)

    # Assert
    self.assertEqual(expected_result, result)
```

7. **Write minimal Production Code (GREEN)**
   Write only the code necessary to pass the test – the Golden Rule! Don't anticipate future requirements. Commit code to Git.

```
class HighestNumberFinder:
  def find_highest_number(self, numbers):
    """
    Returns the highest number from the list.
    Assumes the list is not empty and contains
    only integers.
    """
    return numbers[0]
```

8. **Write your 2nd Test – Return First Item (RED)**
   Write a new test that returns the first item from a list of two numbers in descending order.
    - Name the test: test_find_highest_in_list_of_two_descending_expect_first_element
    - If the input is [13, 4] it will return 13.

```
  def
test_find_highest_in_list_of_two_descending_expect_first_element(self):
    # Arrange
    numbers = [13, 4]
    expected_result = 13
    cut = HighestNumberFinder()

    # Act
    result = cut.find_highest_number(numbers)

    # Assert
```

9. **Write minimal Production Code (GREEN)**
   In this instance, no extra code is required as the existing logic still works.
   - Ensure test passes
   - Commit code to Git.

10. **Write your 3rd Test - Two Items (Ascending), return Second Item (RED)**
    Write a new test that returns the second item from a list of two numbers in ascending order.
    - Name the test:
      test_find_highest_in_list_of_two_ascending_expect_second_element
      If the input is [7, 13] it will return 13.

```
def test_find_highest_in_list_of_two_ascending_expect_second_element(self):
    # Arrange
    numbers = [7, 13]
    expected_result = 13
    cut = HighestNumberFinder()

    # Act
    result = cut.find_highest_number(numbers)

    # Assert
    self.assertEqual(result, expected_result)
```

11. **Write minimal Production Code (GREEN)**
    Update the production code to return the highest number in both cases of a list of two numbers in ascending or descending order.

```
class HighestNumberFinder:
    def find_highest_number(self, numbers):
        """
        Returns the highest number from the list of at most 2 elements.
        """
        if not numbers:
            return None

        highest_so_far = numbers[0]

        if len(numbers) > 1 and numbers[1] > highest_so_far:
            highest_so_far = numbers[1]

        return highest_so_far
```

- Ensure test passes
- Commit code to Git

12. **Refactor Code - Optional.**
   We now see that the current solution doesn't generalise will as it only handles two elements. Let's refactor to handle any list of numbers.
   - Use a Pythonic solution if possible (e.g. built-in max() function)
   - Re-Run ALL tests to confirm no Regression.
   - Commit your refactored code.

```
def find_highest_number(self, numbers):
  return max(numbers)
```

13. **Ongoing Development**
   For each new requirement:
   - Write a FAILING test (**RED**)
   - Write just enough production code to make it PASS (**GREEN**)
   - REFACTOR if necessary (**REFACTOR**)
   - RE-run ALL Tests
   - Commit Code