# QLC-2) - Find the Highest Number Extension.

## Objective

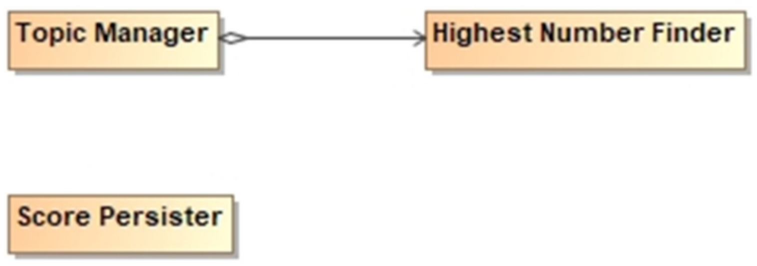
In this lab, you will develop a simple Python class called TopicManager using Test-Driven Development (TDD). You will write tests first, followed by the minimal production code needed to make those tests pass. Each step reflects the RED → GREEN → REFACTOR cycle of TDD. We will also follow a common pattern 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 an list of integers. [DONE]
- A class to find the highest score for a topic.
- A class to write the topic and score to a file on the disk.



You are going to follow a **TDD** approach to find the highest score for a series of topics.

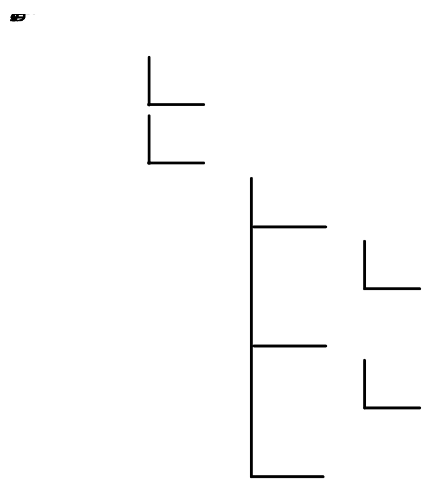
## Given the following specification

If the input is:

- [“Physics”, [56, 67, 45, 89]], the result should be [“Physics”, 89]
- [] the result should be []
- [[“Physics”, [56, 67, 45, 89]], [“Art”, [87, 66, 78]], the result should be [[“Physics”, 89], [“Art”, 87]]
- [[“Physics”, [56, 67, 45, 89]], [“Art”, [87, 66, 78]], [“Comp Sci”, [45, 88, 97, 56]]], the result should be [[“Physics”, 89],[“Art”, 87],[“Comp Sci”, 97]]

Steps

1. **Create your Application Folder**
   Inside HighestNumberServices, create a new root folder for this application called **topic-manager** with the following structure:



2. **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.
3. **Designate main.py as the launcher**
   Place main.py in the topic-manager folder (above app and tests). This script will serve as the launch point for running your application and as the root for test discovery.
4. **Create your first test file**
   Create a test script named test_topic_manager.py inside the tests folder.
   In this file:
   - Define a class called TestTopicManager (following PEP 8 naming conventions for test classes).
   - Ensure it inherits from unittest.TestCase, which provides the framework for writing unit tests in Python.
5. **Write your 1st test (RED)**
   Write your first test to find the highest score in an empty list.
   - It should return an empty list.
   - Your first test method should be named something like:

     def test_find_highest_score_in_empty_list_returns_empty_list(self):

```
class TestTopicManager(unittest.TestCase):
  def test_find_highest_score_in_empty_array_returns_empty_list(self):
    # Arrange
    scores = []                                         # Line 4
    cut = TopicManager()
    expected_result = []                                # Line 6

    # Act
    result = cut.find_topic_high_scores(scores)         # Line 9

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

6. **Write minimal Production Code (GREEN)**
   Lines 4, 6, and 9 provide enough context to begin shaping our design. Based on the requirements, we aim to pass an array of topics along with their associated scores into the find_topic_high_scores method. This method should return the highest score for each topic—represented as TopicTopScore. Each item in the input array can be referred to as a TopicScores object.

   Let's refactor the code to align with this design approach.

```
class TopicManager:
  def find_topic_high_scores(self, topic_scores_list):
    top_scores = []
    return top_scores
```

7. **Write Application dependencies**
   We now need to complete the initial parts of the implementation code to pass the test:
   - Create a topic.py file with a class called Topic with the methods get_topic_name and get_score:

```
class Topic:
  def __init__(self, topic_name: str, score: int):
    self._topic_name = topic_name
    self._score = score

  def get_topic_name(self) -> str:
    return self._topic_name
```

   - Create a topic_top_score.py file with an empty class called TopicTopScore:

```
class TopicTopScore:
  pass
```

- Create a topic_top_score.py file with an empty class called TopicScore:

```
class TopicScore:
  pass
```

8. **Write your 2nd Test – Return First Item (RED)**
   Write a new test that returns the highest score for one topic from one topic list with multiple scores.
   - Name the test: test_find_highest_score_with_list_of_one_returns_list_of_one
   - If the input is [“Physics”, [56, 67, 45, 89], it will return [Physics, [89]]

```
def test_find_highest_score_with_list_of_one_returns_list_of_one(self):
  # Arrange
  scores = [56, 67, 45, 89]
  topic_name = "Physics"
  topic_scores = [TopicScores(topic_name, scores)]

  cut = TopicManager()
  expected_result = [TopicTopScore(topic_name, 89)]

  # Act
  result = cut.find_topic_high_scores(topic_scores)

  # Assert
  self.assertEqual(result[0].get_topic_name(), expected_result[0].get_topic_name())
  self.assertEqual(result[0].get_top_score(), expected_result[0].get_top_score())
```

9. **Write minimal Production Code (GREEN)**
   Update the topic_manager.py file to pass the 2nd test.

```
class TopicManager:
  def __init__(self):
    self._highest_number_finder = HighestNumberFinder()  # Line 3 – Tightly Coupled!

  def find_topic_high_scores(self, topic_scores_list):
    top_scores = []

    if len(topic_scores_list) == 1:
      ts = topic_scores_list[0]
      top_score = self._highest_number_finder.find_highest_number(ts.get_scores())
      top_scores.append(TopicTopScore(ts.get_topic_name(), top_score))

    return top_scores
```

Additionally, update the following:

- TopicScores class

```
class TopicScores:
  def __init__(self, topic_name, scores):
    self._topic_name = topic_name
    self._scores = scores

  def get_topic_name(self):
    return self._topic_name

  def get_scores(self):
    return self._scores
```

Additionally, update the following:

- TopicScores class

```
class TopicTopScore:
  def __init__(self, topic_name, score):
    self._topic_name = topic_name
    self._top_score = score

  def get_topic_name(self):
    return self._topic_name

  def get_top_score(self):
    return self._top_score
```

10. **REFACTOR tightly coupled code**

You may have noticed on Line 3 of TopicManager.py in Step 9 has a comment, in bold so you notice it – like an elephant in the room! In fact this is referred to as an **elephant** because it's the source of **tight coupling**. You cannot write a test for TopicManager without also executing HighestNumberFinder – which makes isolated testing difficult and reduces modularity.

It would be more Pythonic, and testable to pass HighestNumberFinder into TopicManager and also adheres to the SOLID principle – where the 'S' indicates each class/module should have a single, well-defined responsibility. That is, it should focus on one specific task and encapsulate all required logic.

Let's fix this **code smell** by injecting HighestNumberFinder into TopicManager when TopicManager is created (__init__).

```
class TopicManager:
  def __init__(self, highest_number_finder=None):
    if highest_number_finder is None:
      highest_number_finder = HighestNumberFinder()
    self._highest_number_finder = highest_number_finder

  def find_topic_high_scores(self, topic_scores_list):
    top_scores = []

    if len(topic_scores_list) == 1:
      ts = topic_scores_list[0]
      top_score = self._highest_number_finder.find_highest_number(ts.get_scores())
      top_scores.append(TopicTopScore(ts.get_topic_name(), top_score))

    return top_scores
```

This now let's TEST code or other consumers INJECT a different implementation of HighestNumberFinder (e.g. a stub or a mock) and has a default backup.

- Re-Run ALL tests to confirm no Regression.
- Commit your refactored code.

Tests are a great way of identifying code smells. Highly coupled code leads to untestable code.

You want to test one class and one class only. The previous version of the TopicManager dragged in HighestNumberFinder . So inadvertently you were testing that class as well. This will become clearer as we continue to work through the TopicManager

## QLC-2.2) – Working with Stubs

### Overview

A **Stub** is a type of **Test Double** (from Stunt Doubles in the movies), designed to help tests focus solely on the behaviour of the Class Under Test (CUT), rather than its dependencies. In unit testing, it's essential to maintain a controlled and predictable environment—Test Doubles provide this stability.

A **Stub method** returns *canned* results—predefined values that do not rely on complex logic or external systems. These results can be:

- A specific fixed value,
- A range of values,
- Or even a generic/default value.

Similarly, the method parameters can be tightly defined, span a range, or accept any input. Stubs ensure your test outcomes remain consistent and isolated from unpredictable behaviour in dependent components.

11. **Write your 3rd Test - Two Items (Ascending), return Second Item (RED)**
    Write a new test that finds the highest score with one topic using a stub.
    - Name the test:
      test_find_highest_score_with_one_topic_using_stub

- The find_highest_number method now returns a predefined result/stub.

12. W

```
class StubHighestNumberFinder:
  def find_highest_number(self, numbers):
    return 89  # Predefined, canned result

class TestTopicManager(unittest.TestCase):
  def test_find_highest_score_with_one_topic_using_stub(self):
    # Arrange
    scores = [56, 67, 45, 89]
    topic_name = "Physics"
    topic_scores_list = [TopicScores(topic_name, scores)]

    hnf_stub = StubHighestNumberFinder()
    cut = TopicManager(hnf_stub)

    expected_result = [TopicTopScore(topic_name, 89)]

    # Act
    result = cut.find_topic_high_scores(topic_scores_list)

    # Assert
    self.assertEqual(result[0].get_topic_name(),
expected_result[0].get_topic_name())
    self.assertEqual(result[0].get_top_score(), expected_result[0].get_top_score())
```

**rite minimal Production Code (GREEN)**

There should be no need to modify the application production code, from the previous REFACTOR, for the tests to pass.

- Ensure test passes
- Commit code to Git

13. **Refactor Code - Optional.**

No refactoring required at this stage.

- Re-Run ALL tests to confirm no Regression.
- Commit code to Git, if not done in previous step.

You should now be confident to write more tests to handle the last two requirements:

## QLC-2.3) – Limitations of Stubs, complete this requirement

**Lab Exercise**

Write a test for this requirement using a stub, and production code to pass the test. This requirement is finding highest scores with list of many topics and lists of scores.

- [[“Physics”, [56, 67, 45, 89]], [“Art”, [87, 66, 78]], the result should be [[“Physics”, 89], [“Art”, 87]]

- Only ONE of the tests passes. Why?

## QLC-2.4) – Mocks, complete the last requirement

### Overview

In the previous exercise using stubs, only one of these tests work because if the nature of the stub being used – specifically, your StubHighestNumberFinder always returns the same “canned” value, e.g. 89 regardless of the input scores.

```
class StubHighestNumberFinder:
  def find_highest_number(self, numbers):
    return 89  # Predefined, canned result
```

So, in this test, we have deliberately hardcoded 89 as the expected score for all topics to match the stub.

```
    expected_result = [
      TopicTopScore("Physics", 89),
      TopicTopScore("Art", 89),
      TopicTopScore("Comp Sci", 89)
    ]
```

In this instance, if you are trying to verify real calculations, like:

- Highest in [87, 66, 78] should be 87, not 89
- Highest in [45, 88, 97, 56] should be 97, not 89

Then you should replace the stub with the real HighestNumberFinder or a **mock** that returns appropriate values per topic.

**Lab Exercise**

Write a test for this requirement using mocks, and production code to pass the test.

- [[“Physics”, [56, 67, 45, 89]], [“Art”, [87, 66, 78]], the result should be [[“Physics”, 89], [“Art”, 87]]
- RE-run ALL Tests
- Commit Code