5 - Chessington (Python)

There's a well-documented shortage of Chess applications worldwide. Your task is to build something to fill this gap, with the snappy title of Chessington.

Learning goals

The goals of this exercise are to learn about:

- Unit testing
- Test Driven Development (TDD)

Background

Software is a slippery beast. Development sometimes feels like a vigorous game of Whack-a-Rat™, with bugs popping up as swiftly as they are squashed. Today's exercise is going to teach you some strategies for preventing bugs and keeping quality high.

There are many ways to check that your software is correct. This simplest is to run your program and test every feature to make sure that it's behaving correctly. The problem comes when you make a change to part of your program. Re-testing your software after every change you make is time-consuming and expensive. It's also mind-numbingly boring. Developers use Unit Testing as one strategy to reduce the amount of testing required after making changes to an application.

Unit Testing is the practice of writing automated tests which can verify the behaviour of your code. Each unit test should check a single fact about your program. Keeping unit tests small and focused means that when one fails, you know exactly which functionality has broken. A unit test normally contains three parts:

- **Arrange** - create the class you're going to test, and any dependencies.
- **Act** - do the thing which you want to check is correct.
- **Assert** - check that the right thing happened.

This may sound pretty abstract. It will become clearer once you start working on this exercise.

Setup instructions

Start off by forking the Chessington starter project on GitHub: https://github.com/CorndelWithSoftwire/chessington-python Then clone this fork locally:

```
cd \Work\Training
git clone git@github.com:YourNameHere/chessington-python.git
```

After cloning the project, you'll need to install its dependencies - a testing framework, and a GUI framework. Install Poetry (a tool for managing Python codebases) and then run the command `poetry install` from the root directory of the repository.

Open the project in your editor, then have a look around the project. You'll see a unit test file under the chessington/tests directory. This contains three unit tests. Let's run them and see what happens. To do this, run `poetry run pytest tests` from the command line. You should see some output containing the text `3 passed in 0.06 seconds`, or something similar. This is telling you that `pytest` has successfully run the three tests.

Now run the chess program by running the command `poetry run start` from your terminal. You should see a beautiful Chess Board appear, full of pieces. Click on a white piece - it should be highlighted in blue, and squares it can move to will be in purple. Except... Oops, none of the logic around which piece moves are valid has been implemented yet. That's going to be your job! Read on...

The Red-Green-Refactor challenge

Test Driven Development (TDD) is a process for writing unit tests in lock-step with code. Here's the mantra:

- **Red**. We are going to write a test (or tests) which fail(s).
- **Green**. We are going to write **THE SIMPLEST CODE THAT MAKES THE TEST PASS**.
- **Refactor**. Once all of our tests pass, we will try to refactor our code to make it beautiful, without breaking any tests.

Writing good unit tests is hard, so we've written some for you. This will allow you to get used to the Red-Green-Refactor cycle. Once you've got

the hang of this, you can start writing your own tests.

Testing frameworks

You might be wondering what makes test code different from normal code, and how `pytest` knows how to run it. The answer is that `pytest` will use some very simple rules to determine which code is test code:

- It will only look at files in the directory that it is told to run tests from - in the test command above, we tell it to run all tests in the `tests` dir ectory
- It will look for files in that directory which match the pattern `test_*.py` or `*_test.py`
- Within those files, it will look for functions or methods matching thep pattern `test_*` or `*_test`

There are several testing frameworks for Python projects, including the module `unittest` from the standard library, but `pytest` is one of the most widely used and most popular.

You'll also have noticed that all the tests end with `assert ...` . This is the assertion, the part of the test that makes it useful. If we didn't have this, `pytest` would just run the code in the test method to the end and mark it as successful, without actually testing anything! The `assert` lines are how we tell `pytest` that the test should only pass if a certain condition is met.

Steps to repeat

First, grab some unit tests:

```
git cherry-pick red-1
```

Run your unit tests, and verify that some of them fail. (This is the Red bit).

Find the failing test(s) in the test runner. Read them, and make sure you understand what they are testing.

Once you understand your test, write **THE SIMPLEST CODE THAT MAKES THE TEST PASS**. (I will continue with **Bold** and CAPS because this is important).

Note that for the whole of this exercise (apart from one of the stretch goals), you won't need to change anything in the `chessington.ui` module - all the game functionality is contained in `chessington.engine`.

Once your tests are green, see if you can improve things by refactoring your code to make it nicer (without changing its functionality!). This may not always be possible, but Refactor is an optional step so don't sweat it.

When you're happy, run the program and bask in the glory of your new functionality. Then commit your changes with a meaningful comment, and then grab the next set of failing unit tests:

```
git commit -am "Made prawns move, or something..."
git cherry-pick red-2
```

Keep repeating this cycle up to `red-5`. Remember to run all the tests before committing (not just the ones that were failing) - you want to always be sure that your new functionality hasn't broken any existing behaviour.

At this point you should have pawns moving and capturing correctly. But what about the rest of the pieces?

Writing your own tests

Now it's your turn to start writing your own tests!

Choose another piece and write a test for it - from looking in the pawn tests you should have all the ingredients you need to get started on your own.

Now run your test, see it fail, and make it pass. Continue this (remembering to commit frequently) until you have another piece moving correctly. Continue until all the pieces are done.

While you're doing this, keep a look out for opportunities for refactoring - this isn't limited to the game code you've written - it might turn out that as your tests grow, there is some common functionality that should be extracted between individual tests, or between the test classes. Also, you'll definitely need to add to, and change, the `Board` and `Piece` classes as you introduce more functionality, so don't be afraid to refactor the starting

code if necessary.

Stretch goals

More rules

Implementing the moves for each individual piece will only take you part of the way to the full game of chess. When all of your pieces move correctly, try to implement some of this functionality:

- En Passant
- Castling
- Pawn Promotion
- Check and Check Mate
- Stalemate

Don't forget to write a failing test first!

Make it look good!

See if you can make any changes or additions in `chessington.ui` to improve or change the look of the board. Is there any other information the game window could be showing?

Develop a chess AI

Seriously, how hard can it be? See if you can write a simple AI which plays the Black pieces. Start by picking a valid move at random.