"Testing 1, 2, 3" Exercise

> Prerequisites:
>
> - Software Maintenance and Quality Control
> - The `pytest` Package

Learning Objectives

- Practice writing automated tests to verify your program functions as expected.

Instructions

Exercise 1: Testing Single-File Python Modules

To setup this first exercise, create a new directory on your Desktop called "testing-123" and navigate there from your command line, then create files called "my_script.py" and "my_test.py" and place inside the following contents, respectively:

```
# testing-123/my_script.py

def enlarge(i):
    return i * 100
```

```
# testing-123/my_test.py

from my_script import enlarge

def test_enlarge():
    result = enlarge(3)
    assert result == 300
```

> NOTE: the testing function's name is prefixed with "test_", it directly invokes the function we want to test (e.g. `enlarge()`), and describes expectations for that function's desired behavior

Once you have setup the example files, create and activate a new Python 3.7 virtual environment, then from within the virtual environment: install the Pytest package and use it to run the tests:

```
# navigate to the right directory:
cd testing-123/
```

```
# configure a virtual environment:
conda create -n testing-123-env python=3.7 # (first time only)
conda activate testing-123-env

# install pytest package (first time only)
pip install pytest

# run the tests:
pytest #> 1 passed in 0.01 seconds
```

Nice!

> NOTE: this will generate some files in a new directory called "__pycache__". These files should be ignored from version control, using a ".gitignore" file!

Exercise 2: Testing Multi-File Python Modules

When testing larger applications, conventions suggest we should move the application and test files into their own separate directories called "app" and "test", respectively.

To set up this exercise, first move your application file into an "app" directory:

```
# testing-123/app/my_script.py

def enlarge(i):
    return i * 100
```

Then, move your test file into a "test" directory, and slightly modify its `import` statement to reflect the application's new location (e.g. `app.my_script`):

```
# testing-123/test/my_script_test.py

from app.my_script import enlarge

def test_enlarge():
    result = enlarge(3)
    assert result == 300
```

Once your repository structure looks like this, if you were to try to run tests again, you'd run into an error `ModuleNotFoundError: No module named 'app'`. We need to indicate that scripts inside the "app" directory should be able to be loaded / imported by files in other directories.

In some cases this can be achieved by adding a special file called "__init__.py" to the "app" directory. But for testing purposes, the professor recommends you also add a special file called "conftest.py" to the repository's root

directory. Even if the contents of these files are empty, it helps the Pytest package locate the proper files.

Once you have finished setting up this example, including the "app/__init__.py" and "conftest.py" files, run the tests again:

```
pytest #> 1 passed in 0.01 seconds
```

Exercise 3: Testing Executable Scripts

So far the Python file we are testing only contains isolated functions, and doesn't do anything when we invoke it from the command-line. But what if we wanted to test the functions inside a Python file that is also an executable script in its own right?

Modify the "my_script.py" such that when it is invoked it will use its own `enlarge()` function to process user inputs:

```
def enlarge(i):
    return i * 100

original_number = int(input("Please select a number to be enlarged (e.g.
400): "))
print("You chose:", original_number)

bigger_number = enlarge(original_number)
print("Enlarged number is:", bigger_number)
```

And invoke it as a script:

```
python app/my_script.py
# OR:
# python -m app.my_script
```

Great! At this time, you should be able to invoke the script successfully, but when you try to re-run tests, you will see an error:

```
pytest #> OSError: reading from stdin while output is captured
```

When the test file imports the code from the script, it will execute all code in the script's global scope, including the new code which asks for a user input. But it doesn't make sense for our automated test to ask a user for inputs, as there is no user involved in the process. So we need a way to isolate the `enlarge()` function's definition from its

invocation. We can use a special convention (`if __name__ == "__main__"`) to check whether the file is being invoked from the command-line, or is being loaded from / imported by another file. This allows us to distinguish between what should happen in each case, and prevents certain functionality from being executed when the script is imported as a module. For more details, see also Custom Modules in Python.

Let's make that final change now:

```
def enlarge(i):
    return i * 100

# "if this script is run from the command-line, then ..."
if __name__ == "__main__":
    original_number = int(input("Please select a number to be enlarged (e.g.
400): "))
    print("You chose:", original_number)

    bigger_number = enlarge(original_number)
    print("Enlarged number is:", bigger_number)
```

After making this small change, you should be able to successfully invoke the script from the command-line, as well as import its functionality into tests:

```
python app/my_script.py
# OR:
# python -m app.my_script
```

```
pytest #> 1 passed in 0.01 seconds
```

Nice job! You're testing like a pro!