Lab 2 : Integration and Unit Tests

The best way to learn is by doing. You now understand all the essential practices of continuous integration, so it's time to get your hands dirty and create the whole chain of steps necessary to use CI. This chain is often called a CI **pipeline**.

This is a hands-on tutorial, so fire up your editor (VSCode, PyCharm, …) and get ready to work through these steps as you read!

Problem Definition

Remember, your focus here is adding a new tool to your utility belt, continuous integration. For this example, the Python code itself will be straightforward. You want to spend the bulk of your time internalizing the steps of building a pipeline, instead of writing complicated code.

Imagine your team is working on a simple calculator app. Your task is to write a library of basic mathematical functions: addition, subtraction, multiplication, and division. You don't care about the actual application, because that's what your peers will be developing, using functions from your library.

Task 1 - Set Up a Working Environment

It is highly recommended to set up an environment for each project you are working on. So go ahead and create a virtual environment somewhere within the project directory.

```
# Create virtual environment
python3 -m venv venv

# Activate virtual environment (Mac and Linux)
source venv/bin/activate

# For Windows
venv\Scripts\activate
```

Task 2 – Simple Calculator

Within the file utils.py, create a function for each basic algebraic operation: addition, subtraction, multiplication and division.

```
def add(n1, n2):
    return n1 + n2
```

This is a bare-bones example containing one of the four functions you will writing.

Task 3 – Writing Unit Tests

You will test your code in two steps. The first step involves **linting**—running a program, called a linter, to analyze code for potential errors. flake8 is commonly used to check if your code conforms to the standard Python coding style. Linting makes sure your code is easy to read for the rest of the Python community.

```
# Run linter
flake8 --statistics --exclude venv –-ignore=E501,F401
```

The `--statistics` option gives you an overview of how many times a particular error happened.

If you have PEP8 violations, fix them and rerun the linter.

Now it's time to write the tests. Create a file called `test_algebraic_functions.py` in the tests/unit/ directory and copy the following code:

```
"""
Unit tests for the calculator library
"""

import os, sys

# Add project root to Python path
project_root = os.path.abspath(os.path.join(os.path.dirname(__file__), "../.."))
sys.path.insert(0, project_root)

from src.utils import add, subtract

class TestCalculator:

    def test_addition(self):
        assert 4 == add(2, 2)

    def test_subtraction(self):
        assert 2 == subtract(4, 2)
```

These tests make sure that our code works as expected. Implement as many assertions you think are important for potential code misuse. Also, implement the remaining tests for the other functions

Once you've implemented all the tests you needed. Run the following command:

```
$ pytest -v --cov
```

`pytest` is excellent at test discovery. Because you have a file with the prefix `test`, `pytest` knows it will contain unit tests for it to run. The same principles apply to the class and method names inside the file.

The `-v` flag gives you a nicer output, telling you which tests passed and which failed. In our case, both tests passed. The `--cov` flag makes sure `pytest-cov` runs and gives you a code coverage report for `utils.py`

You have completed the preparations.

Task 5 – Let's make it a bit bigger

Now that you have your four algebraic operations. Add a file calculator.py under src folder. Within the newly created python script, do the following:

- Ask the user to input his choice as a number (1,2,3 or 4 depending on the operation)
- Then depending on the user choice, ask the user to input two numbers. These numbers will be the parameters for the algebraic function.
- Check if the user wants another calculation or not.
- Make sure to handle invalid user inputs.
- Can you think of any tests to implement? Integration tests?