Learn about coverage

Coverage

1. In your cicd repository, create a branch `lesson_6`, we will use it to learn and you should not merge it.
2. Read about coverage in the docs for pytest-cov
3. Install `pytest-cov` and add it to `requirements.txt`
4. Calculate coverage on your cicd project with `pytest --cov=shop_app`
5. We could also specify the tests folder as before `pytest --cov=shop_app tests`
6. Pytest-cov has more configuraion
    1. Read about and test different `--cov-report=type`
    2. Read about and test the `--cov-fail-under=MIN`
7. To test the `--cov-branch` we must add code to our project that contains a branch.
    1. Add a file `lottery.py` to your `shop_app`folder with the code:

    ```
    def lucky(x):
        if x > 10:
            return "WON"
        return "SORRY"
    ```

    2. Run `pytest --cov=shop_app tests` read the result for `Stmts`, `Miss` and `Cover`
    3. Run `pytest --cov=shop_app --cov-branch tests` now you should get two more columns `Branch` and `BrPart`, you should also notice that the `Cover` percentage has changed
8. We will now add dummy file to learn about omit config. Add `file_to_exclude.py` in your `shop_app` folder.
    1. Run pytest with coverage and find the file in your coverage report
    2. Add a file `.coveragerc` to your project root with the content:

    ```
    [run]
    omit = shop_app/file_to_exclude.py
    ```

    3. What happens when you run coverage again?
    4. What happens if you change omit to `shop_app/*`
9. Creata a file `pytest.ini` in your root project folder, with the content:

```
[pytest]
addopts = --cov-branch --cov-report html
```

10. Run `pytest --cov=shop_app tests` what happens?
11. Run `pytest tests` what happens?
12. Change your config file to `--cov-report term-missing`, what is added?
13. Add another `--cov-report=xml`to your config file, this should print both in terminal and to a file `coverage.xml`.
14. Add the generated files and folder to your `.gitignore` file, make sure your don't commit any generated files. You should still be able to commit config file.
15. Install `coverage-gutter` in vscode, make sure it shows the coverage.
16. Write tests to improve your coverage, rerun `pytest --cov=shop_app tests` after each added test and read the output.

Pytest Raises

1. Read more about pytest raises
2. Create a function that uses try/except on a specific error.

```
def division(x, y):
    try:
        return x/y
    except ZeroDivisionError:
        print("Can't division with zero")
```

3. Run pytest with coverage, check the calculated coverage.
4. Write a test for the non failing branch
5. Run pytest with coverage, check the calculated coverage.
6. Write a test for the except branch
7. Run pytest with coverage, check the calculated coverage.
8. Write a `pytest.raises` that test i.e division by a `str`

Pytest fixture

1. Read more about fixtures, copy the fruit example and run it, modify, understand. Try to change args, function names etc. try to break and fix it.
2. Create a class that requires setup, e.g Car() and write tests

3. Create a fixture that setup the class

Pytest Duration

Unit tests should be fast, so to catch slow bad tests there are some helping commands.

1. Create a test named `test_slow.py` with the code:

```
import time

def test():
    time.sleep(1)
    assert True
```

2. Run your pytest with the options `--durations=5 --durations-min=1`
3. What happens? How can you use this?
4. Copy the test file multiple times and name them `test_slow_one.py`, `test_slow_two.py`..
5. Adjust the sleep time to 1, 2
6. Run pytest again

Parameterize

1. Read more about parameterize, run and understand the examples
2. Write your own test that uses parameterize

Linting and static code analysis

Pylint

1. Install pylint

```
pip install pylint==2.15.5
```

2. Generate config

```
# Linux & mac
pylint --generate-rcfile > .pylintrc
```

3. Run pylint on all .py files in git repository

```
# Linux & mac
pylint $(git ls-files '*.py')
```

4. Try to fix linting errors in the code, you can also ignore warnings if they are hard to fix.

5. Configure pylint in Visual Studio Code docs linting

Autopep8

1. Install autopep8

```
pip install autopep8
```

2. Create a messy.py file from the autopep usage example:

```
import math, sys;

def example1():
    ####This is a long comment. This should be wrapped to fit within
72 characters.
    some_tuple=(   1,2, 3,'a'  );
    some_variable={'long':'Long code lines should be wrapped within 79
characters.',
    'other':[math.pi, 100,200,300,9876543210,'This is a long string
that goes on'],
    'more':{'inner':'This whole logical line should be
wrapped.',some_tuple:[1,
    20,300,40000,500000000,60000000000000000]}}
    return (some_tuple, some_variable)
def example2(): return {'has_key() is
deprecated':True}.has_key({'f':2}.has_key(''));
class Example3(   object ):
    def __init__    ( self, bar ):
    #Comments should have a space after the hash.
    if bar : bar+=1;  bar=bar* bar   ; return bar
    else:
                    some_string = """
                    Indentation in multiline strings should not be
touched.
Only actual code should be reindented.
"""
                    return (sys.path, some_string)
```

3. Test autopep8 on the file, what is the difference?

```
autopep8 messy.py > standard
autopep8 -a -a messy.py > aggressive

diff standard aggressive
```

4. Configure autopep8 in Visual Studio Code editing formatting

Pre-commit hook

Pre-commit is a framework that helps you to check the code before its committed or pushed the central code repository. It has multiple framework hooks.

1. Install either in the system or with

```
# https://pre-commit.com/#installation
pip install pre-commit

# Test the installation
pre-commit --version
```

2. Generate a config in the project root

```
# To generate a sample config
pre-commit sample-config

# On linux & mac generate and create  config file
pre-commit sample-config > .pre-commit-config.yaml
```

3. Install pre-commit hooks, this will enable them in your local git.

```
 pre-commit install
```

4. Pre-commit will only check new file changes, so it's a good idea to fix all existing files when introducing new hooks.

```
pre-commit run --all-files
# Remember to commit fixes
```

5. [Optional] you can update your hooks with

```
pre-commit autoupdate
```

6. Add Pylint to pre-commit hooks

```
```

```
-   repo: https://github.com/PyCQA/pylint
    rev: v2.15.5
    hooks:
    -   id: pylint
        language: system
        args:
        - --max-line-length=120
        - --output-format=colorized
        - --errors-only
```

7. It not an good idea to duplicate config, so the max-line-length=120 should be moved to the pylint configuration file

8. EXTRA Configure autopep8 in pre-commit

Pytest Mock

Patch

1. Install pytest-mock and add it to your requirements.txt

2. First we will test the official examples

```
import os

def test_foo(mocker):
    # all valid calls
    mocker.patch('os.remove')
    mocker.patch.object(os, 'listdir', autospec=True)
    mocked_isfile = mocker.patch('os.path.isfile')
```

3. Read about autospeccing

4. Add the line `os.listdir.whatever()` and run the test. Test with `autospec=True` and `autospec=False` what happens? Remove the line when done.

5. *EXTRA* to get extra help typing you can use mypy with type annotation.

```
from pytest_mock import MockerFixture

def test_foo(mocker: MockerFixture) -> None:
    # all valid calls
    mocker.patch('os.remove')
    mocker.patch.object(os, 'listdir', autospec=True)
    mocked_isfile = mocker.patch('os.path.isfile')
```

6. Create a file `a_dummy.txt` in the project root.

7. Comment out the line `mocker.patch.object(os, 'listdir', autospec=True)` and add assert:

```python
def test_foo(mocker: MockerFixture) -> None:
    # all valid calls
    mocker.patch('os.remove')
    #mocker.patch.object(os, 'listdir', autospec=True)
    mocked_isfile = mocker.patch('os.path.isfile')
    assert os.listdir() == []
```

8. Why does the assert fail? Try to enable the patch of listdir, what happens?
9. We need to set the mock return_value if we want to assert, e.g os.listdir.return_value = [], now it should pass the test.
10. Add a line `os.remove('a_dummy.txt')` and run the test, is the file removed or not?
11. Comment out the line `mocker.patch('os.remove')` and run the test, is the file removed or not?
12. Run the test multiple times, what happens?

EXTRA

EXTRA - Linting - mypy

1. Read more about
    - mypy
    - type hint
2. Enable mypy in visual studio code specific linters