Further Testing Tools

We have written unit tests in pytest that demonstrate correct functioning of production code. These includes edge cases, invalid input, and checks on mutable arguments.

We have shown how these tests can be automatically executed by a continuous integration server, such as gitlab or azure devops.

We've also:

- Written parameterized tests, so that we can use different test values with a single test body.
- Used Mock objects as arguments to the production code, so that the code can be tested using the mock object to simulate the presence of external resources.
- Patched names used by the production code, so that the code can be tested using the patched name instead of the real name, again to simulate the interaction with external resources.

Below, we'll cover some additional testing tools that you may consider using, or encounter when you join a team:

- Linting tools
- The 'tox' testing wrapper
- Code coverage tools
- Mutations

Linting Tools

The word 'Lint' refers to the bits of fluff and dust that can accumulate on clothes. The process of 'linting' refers to the act of finding and removing these bits.

By analogy, software documents can gather various aspects of 'lint' -- superfluous or missing whitespace, and other minor defects that do not cause actual errors but, if left to accumulate, raises the risk of errors creeping in unnoticed.

Another reason for keeping the software 'lint-free' is that, if the respository is maintained in this way, then there is no need for developers to commit changes that simply remove lint (or for the lint-removal changes to make it harder to see the semantically significant changes to the code.) This makes the repository (and the list of changes made to the repository) cleaner and easier for all to use.

There is a Python standard (PEP008) that lists the conventions that should be adopted. This list includes:

- Including two empty lines before each function definition
- Including a space either side of operators, e.g. 'x == y' not 'x==y'

- Removing all unused imports
- Removing variables that have been assigned but never used
- Using lower-case names for objects and functions
- Using title-case names for classes

Linting tools such as flake8 list all contraventions of these conventions (i.e. lists the 'lint' in our source code).
After installing flake8 with pip, it can be run in the root folder of your source code, to produce a list of all warnings and errors.

Challenge: Running flake8 on your local machine

Run flake8 in the folder containing your source code for this module. Correct any errors and warnings, re-running until your source code is 'lint-free'.

Challenge: Running flake8 on your continuous integration server

Add flake8 to your test automation script, on the continuous integration server you are using (e.g. gitlab, github or azure devops).

The 'tox' Testing Wrapper

The python package 'tox' is a testing wrapper that makes it easier to test in different environments (such as different versions of Python).
Instead of testing in the base environment, the testing environments are listed in a configuration file named tox.ini. An example configuration file is shown below:

```
[tox]
envlist = py37,py38
skipsdist = true

[testenv]
deps = pytest
commands = pytest
passenv=PYTHONPATH
```

The final line passes the environment variable PYTHONPATH into the virtual environments that tox creates to run the tests. Assuming that PYTHONPATH is set to include the current working directory ('.'), this will ensure that Python can find the modules to be tested,

Code Coverage Tools

Many teams use code coverage as a metric for measuring how complete the tests are. This provides the team with an overall status indication of the testing code, a view on what elements are
For pytest, a useful code coverage tool is pytest-cov

Challenge: Adding a Code Coverage Estimator to your Test Script

Can you add code coverage to your pytest script? What percentage of your production code is covered by your unit tests?

Testing Using Mutations

If our code has 100% test coverage, then we can use mutations to exhaustively test our code by making random mutations to the production code, to check that the unit tests do fail, for every change in the production code. Hence, mutation testing tests the unit tests. The python tool for this is mutmut, and more details are available here. This is extreme testing!