Introduction and Overview

In this unit, we'll introduce the topic of automated testing, review its objectives (some of which may surprise you), and the types of testing that are most usually automated.

We then give a summary of the subsequent learning units.

What is Automated Testing?

When we are developing software, we check our work in lots of different ways. Some tests require a human to click on a button, or look at a graph on a screen. Other 'automated' tests can be written down as a series of instructions, that can be executed when no humans are present (e.g. in the middle of the night when they are all asleep).

This *series of instructions* is typically referred to as a 'script', though it may actually involve dozens of files scattered across different locations.

So, what types of tests can be automated? It greatly depends on the type of application, and the size of the development team. Some important categories of tests are:

- Unit tests
- Smoke tests
- Integration tests
- Functional tests
- Performance tests
- Acceptance tests

Unit Tests

Unit tests are for isolated components of your code, such as a specific function or an object's method. Normally, one software component will have many unit tests. Out of the various categories of test, unit tests are typically the most straightforward to automate, although there are varying degrees of automation.

In the subsequent units of this module, we'll be concentrating on unit tests (as opposed to the other test categories, listed below).

Smoke Tests

Smoke tests are also called 'sanity tests', designed to give a quick indication of whether the software is operational, before proceeding with more complex tests or demonstrations.

For example, a smoke test could demonstrate a key function such as annotating the images in a 'samples' folder with classification results, or demonstrate that key elements of data can be retrieved from a live web service.

On its entomology: 'The phrase *smoke test* comes from electronic hardware testing. You plug in a new board and turn on the power. If you see smoke coming from the board, turn off the power. You don't have to do any more testing.'*

* *Kaner, Cem; Bach, James; Pettichord, Bret (2002). Lessons Learned in Software Testing. Wiley Computer Publishing. p. 95. ISBN 0-471-08112-4.*

Integration Tests

Integration tests are run to verify that components can be used together. They can be automated tests or run manually.

Functional Tests

Functional tests are designed to show that a specific functional requirement has been met, i.e. that a given story has been correctly designed, implemented and deployed.

Performance tests

Performance tests can show different aspects of the performance, including bench-marking of memory and time for different loading conditions, and accuracy of predictions over a given test set (if this is relevant test for the project in question). They can be scheduled to run automatically, or (more typically) they are run manually.

Acceptance tests

Acceptance tests are used to demonstrate that the specific elements of a contract have been fulfilled and that the client is in a position to accept delivery of these elements. Typically, acceptance tests are not automated.

What are the objectives of automated unit testing?

Typical answers to the above question include the following:

- *To show that the software is doing what it's meant to;*
- *To find problems (bugs, issues, defects...) in the software.*

The above two points are valid, but they aren't the whole story.

One important objective that may be overlooked is:

- *To check that no problems are introduced when the tests are run on in a different environment.*

This can include changes in hardware, operating system, and also different versions of the supporting libraries used by the software we are testing.

The idea of 'checking that all tests pass, after some change' turns out to be a central part of the testing activity. This idea is included in the following objective:

- *To check that a solution to one software problem hasn't created other problems elsewhere in the software.*

Frequently, changes to the software are made that don't include any new features, or bug-fixes, or different libraries. These are changes to *how the software is organized*: making these changes is also known as *refactoring* the software. We can write this objective as:

- *To check that the software still works after it has been refactored.*

As we shall see in the next section, this is probably the most common use-case for re-running the tests: we make some useful change to the code, and then run to tests to check that it all still works.

However, there are some grander objectives that we can include. You may be thinking that, for a small project, you may be able to perform the above checks by looking at the code, and using a few example values, without any formal test framework. That may be true: but what if the software gets bigger and more complicated? The informal approach will fail at some point, and it's actually quite hard to write good tests for previously established software (that doesn't have any tests). So this objective can be written as:

- To allow our software project to scale

This references an interesting point: what makes it hard to write tests for previously untested code? It's because the untested code has typically evolved without testing in mind, meaning that it then becomes harder to locate the appropriate parts of the code to test (and the tests are also more difficult to write). Hence, we can include this as our final objective:

- To guide the architectural development of the software

Collectively, these objectives describe why testing is such an important part of software development. Indeed, software without tests is frequently described as 'legacy code' -- who'd want to be responsible for crafting brand new legacy code?

Below, these objectives are listed together:

- To show that the software is doing what it's meant to;
- To find problems (bugs, issues, defects...) in the software.
- To check that no problems are introduced when the tests are run on in a different environment.
- To check that a solution to one software problem hasn't created other problems elsewhere in the software.
- To check that the software still works after it has been refactored.
- To allow our software project to scale
- To guide the architectural development of the software

In the subsequent learning units, we'll:

1. Review the workflow for *test-driven development*, which is used by many teams
2. Write some simple tests using pytest, a popular test framework for Python
3. Put these tests into a git repository, and write the script that runs these tests automatically, whenever there is an update.
4. Write tests to check that invalid input is handled correctly.
5. Demonstrate three common testing techniques: parameterizing, mocking and patching. These improve the quality of the tests in various ways.
6. Look at some further testing techniques that you may encounter in production systems.