Corndel DevOps Engineering Programme

in association with Softwire

Module 3: Automated Testing

Corndel Digital.

DA6T

Introduction

Automation is central to DevOps, and there are few automation candidates as promising as the testing process. Traditional software testing can take teams weeks, sometimes months, of repetitive work to certify a release. Such processes can have huge costs, both directly and indirectly through the inefficiencies imposed on the wider delivery process. Automated tests streamline this problem, and are a cornerstone of the **Continuous Deployment** philosophy, enabling teams to move quickly and safely in the knowledge that a thorough test suite minimises the risk of broken deployments reaching the live environment.

In this module we'll learn how to write and run automated tests, storing them alongside our application code in source control. We'll discuss the **Testing Pyramid** and how it enables us to test applications at multiple logical levels, reducing the risk of bugs and defects going undetected. We'll gain hands-on experience writing **unit tests** for individual modules and classes, **integration tests** to ensure our components work smoothly together and **end to end tests** to ensure the finished product looks, feels and works as designed.

We'll take automated testing one step further, and introduce the paradigm of **Test-Driven Development**, where failing test suites are a pre-requisite to writing new application code. We'll put this into practice in our exercises, both solo and through Ping-Pong pair programming. We'll learn how a thorough and performant automated test suite is central to modern Agile workflows, reduces the reliance on manual end-to-end testing and is a cornerstone of good DevOps practice.

Table of Contents

Chapter 1

Introduction to Automated Testing

So, you've written some code, and it looks good! It does what you expect, it's easy to read, and it is making your life easier. But you've been asked to test it, and that feels like effort you could be putting into writing more code!

Why do we test software?

We test software to verify that it behaves as we expect it to. You have probably been manually testing the code you have written without even thinking about it - getting code to work involves constant manual testing to ensure that it is doing what it is supposed to. A typical software project will have a lot of people who are dependent on your software behaving as it is intended, so it's really important that it works correctly!

OK, but what about automated tests?

We can manually verify that our code works when we write it, but manual testing provides no future guarantee of your software's behaviour. Most software projects evolve and change a great deal in their lifetime, and as soon as the codebase changes, the last manual test is invalidated. One goal of DevOps is to move towards continuous deployment, and that's only possible if you can be confident your code is working at all times. So we need to do better than manually testing the latest code change.

We automate a test because we want to run the test each time the codebase changes, to ensure that the change has had no impact on existing behaviour. With a comprehensive set of automated tests we can be confident that each version of the code (i.e. each commit to git) that passes the tests is a viable release candidate.

There are lots of reasons the codebase might change:

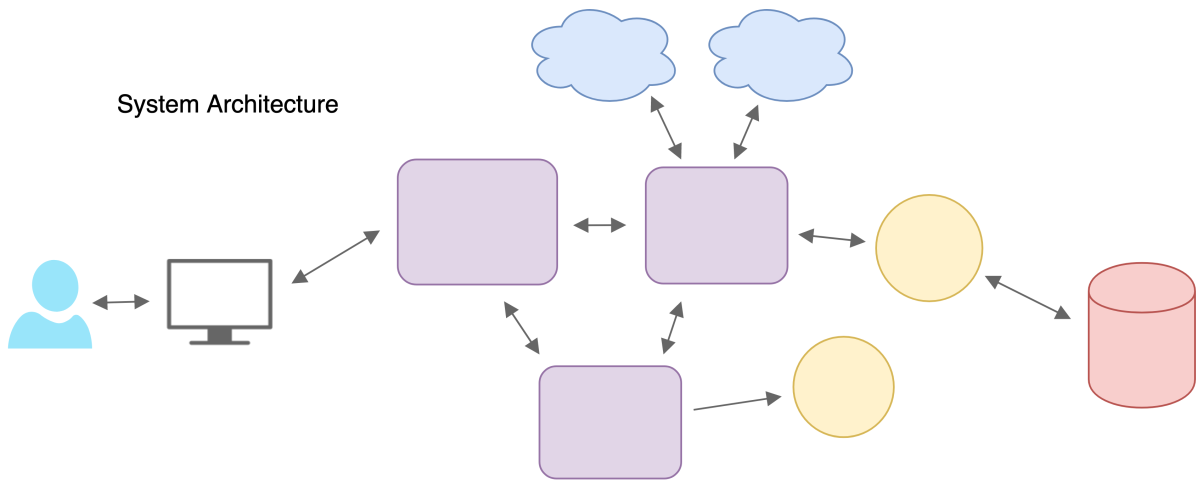
- A new feature is added
- A developer has refactored an area of the codebase
- The configuration of the application has changed
- An API integration has been replaced

In each of these cases, areas of functionality unrelated to the change are preserved and verified by the test cases that have been written.

Of course, if the behaviour of the application changes, the set of automated tests need to change with it. This has the added bonus of making the tests a great candidate for documenting the behaviour of the software, as well as verifying it.

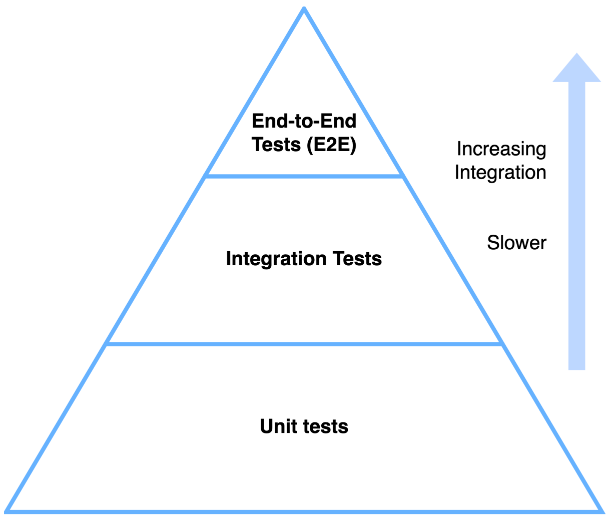
The Test Pyramid

A typical piece of software is like a machine with many moving parts. There will be a user interface, several backend services, databases and connections to external resources, for external a web API used for user authentication. Underlying all this will be some hosting infrastructure - perhaps it's your own server, or a managed hosting service provided by another supplier.



How do you go about writing automated tests in a systematic way? How do you know you've covered everything? You can use tools to report on metrics like code coverage (more on that later), but having tests run every line of code doesn't mean your software is fully tested. Some business requirements and key features aren't determined by a single line of code - they rely on complex interactions between multiple system components. Equally, it's still important to make sure your individual functions and classes behave as expected!

We need a structured approach that tests our system at multiple levels of abstraction from the code. The **Test Pyramid** is a popular way of thinking about this:



A quick look on the internet reveals many versions of the test pyramid, with loads of different names applied to the different tiers! We've taken a common definition here that includes *unit*, *integration* and *end-to-end tests*. Regardless of the specific naming, all versions communicate two simple ideas:

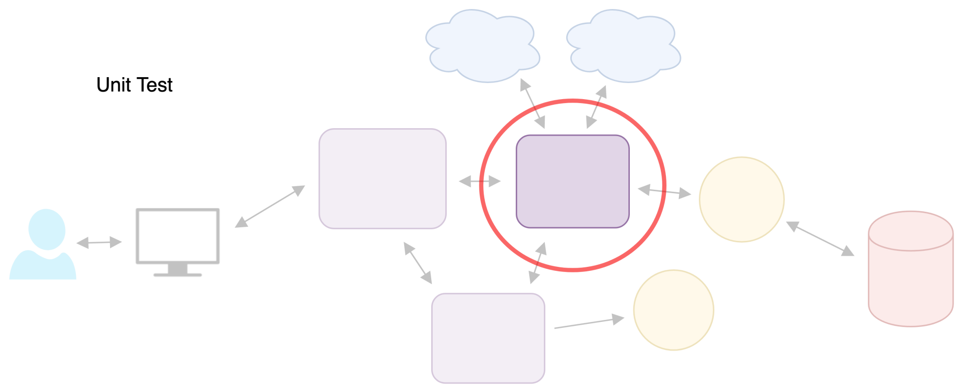
- Higher-level tests build upon lower-level tests. They test at a higher level of abstraction, and shouldn't repeat work done lower in the pyramid.
- When trying to test something, test it as far down this pyramid as possible. You will usually end up with a lot more unit tests than integration tests, and more integration tests than E2E tests.

We can't overstate the importance of testing as low as possible in the pyramid! Unit tests are the most specific and quickest to run. They provide fast feedback that is easy to diagnose and debug. Catching the same bug via end-to-end testing is usually a much more time-consuming task. The test might not be run for weeks, and it can take a while to dig through the test logs and pin-point the underlying issue. Quick and easy test feedback is not just a nice-to-have. It's essential if you want to integrate automated testing into development workflows.

It's important to remember that the Test Pyramid is just a cartoon, and not a hard-and-fast rule! Some projects will require more integration tests than unit tests. Some projects have no UI for a human to test in traditional end-to-end fashion. Nevertheless it's a useful tool to help you think about your testing architecture and the different levels at which you need to test at. Let's dive a bit deeper into each of the testing levels.

Unit Testing

Unit tests will be the most common type of test written by developers. They're usually written alongside (or even before) the code they test. They test small components of the system, typically individual functions or classes. Unit tests are particularly valuable when testing components of the system that involve complicated logic.



Running Tests

In this module we're talking about automating software tests, and for now you'll be kicking off test runs manually, for example by running this command:

```
> pytest
```

You may be thinking: "Ok, but can we automate running the tests, as well as automating the tests themselves?"
This is a good question, and the answer is yes! Automatically running your tests is a very good idea. It can help you find errors more quickly and removes the risk of developers forgetting to run the test suite. There are a few different ways tests are often run automatically, and we'll talk about this further when we come to the topic of "Continuous Integration", but here's a sneak preview.

Watch Modes
Most popular testing frameworks have what's known as a watch mode (or have a plugin that provides this functionality). A test runner in watch mode is a persistent process that watches your source code and test files for changes. When a file changes, the tests are automatically re-run. Many test runners also have optimisations so that only the relevant tests are re-run — this can save a great deal of time if you test suite is slow.
Watch modes are a helpful tool for day-to-day programming when run locally on a development machine. They remove the repetitive task of re-running the test suite, and are particularly useful in development workflows where frequent testing is essential, for example test driven development which we'll dive into later in this Module.

If you're running pytest, you can add watch modes using plugins such as pytest-watch (https://pypi.org/project/pytest-watch/) to provide the watch functionality, and pytest-testmon (https://pypi.org/project/pytest-testmon/) to re-run only what's needed. You can do this using a command like:

```
> ptw --runner "pytest —testmon"
```

Commit Hooks
It's possible to customise VCS workflows to run test suites automatically. If using git, you can leverage the pre-commit and pre-receive commit hooks to run your tests as part of the process. Pre-commit hooks run when a developer commits code to their local repository, and the pre-receive hook runs on the server when new code is pushed. In both cases, you can configure git to abort the process if a test fails.
You can read more about this approach the git documentation: https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks.

CI Pipelines
We'll talk more about CI pipelines in later modules. For now, know that they're a server-side tool that monitors a VCS repository and runs a set of defined steps each time the repository changes. A common set of steps might be:

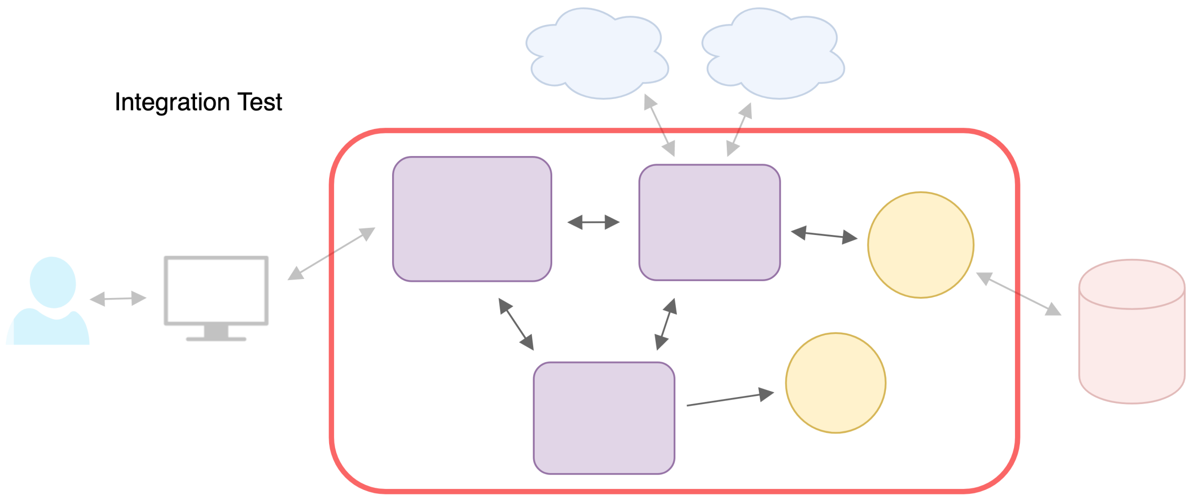
1. Checkout latest version of branch
2. Run unit tests
3. Run integration tests
4. Build deployment artefact
5. Put deployment artefact somewhere

A CI pipeline is a very common way of running tests across a codebase. It ensures a single source of truth, and isn't subject to the machine-to-machine setup differences that can disrupt local developer testing.

Integration Testing

Integration tests cover the *integration* of different components in your system. What is a component? Well that varies. Integrations can check that classes work together, but they also cover much high-level integrations between micro-services, or test database interactions. You'll typically write fewer integration tests than unit tests, as hinted by the width of the pyramid.

Sometimes, **functional tests** are discussed interchangeably with integration tests. We'll use the terms with slightly different meaning, and will come back to this later in the module. For now, be aware that *integration* tests cover the communication between different system components, while *functional* tests look at the behaviour of specific application features, and generally sit somewhere between integration and end-to-end tests.

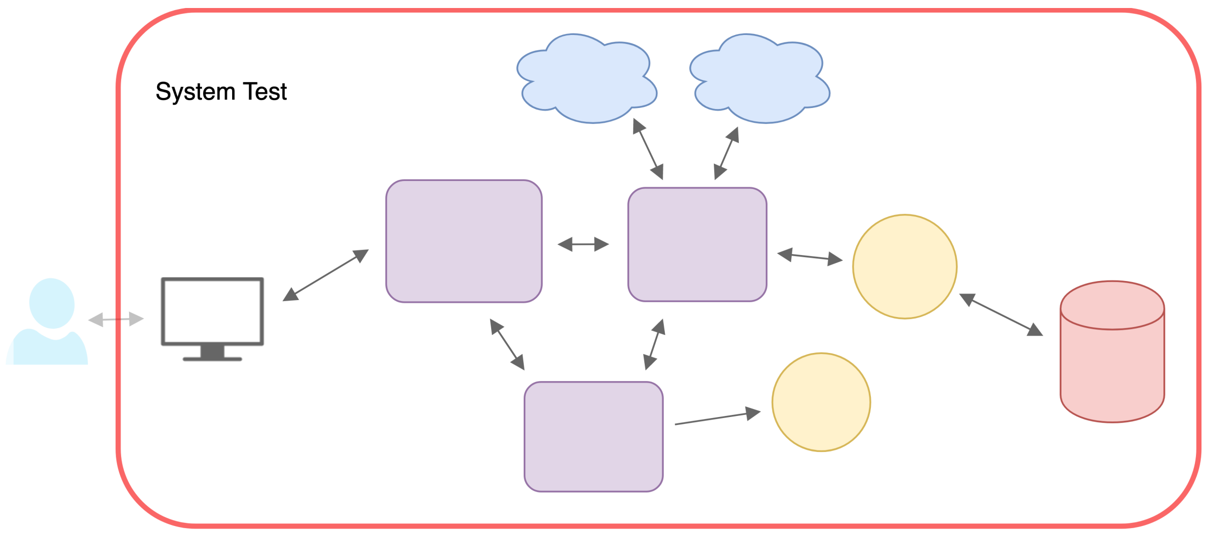


End-To-End Testing

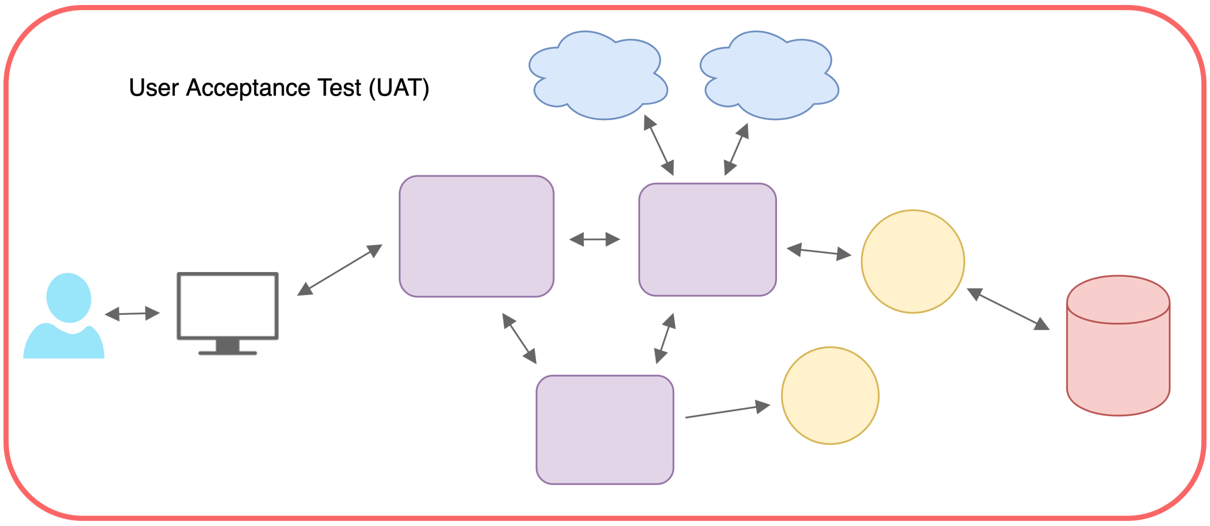
At the top of the pyramid sits **end-to-end** (E2E) testing. E2E testing is about testing the entire software system. The step is often difficult to automate fully, as it tests not only whole-app functionality, but also less technical concerns such as the look-and-feel of a UI, or the ease of onboarding new users.

A key aim of automated testing is to reduce reliance on manual E2E testing. Manual testing is slow, and many of the questions asked (e.g. "Does login work? Do all the buttons render correctly?") can be replaced by automated unit and integration and system tests.

Within E2E testing there are two main types of testing. **System tests** are automated tests of the whole system on production-like infrastructure. They ensure the complete system satisfies technical requirements, and can also be used to check non-functional requirements such as performance, uptime or security.



User Acceptance Testing (UAT) is testing conducted by human users of the software. UAT checks that the system is fit-for-purpose, meets business requirements and provides a way to flag user interface or user experience concerns. The focus here is not on technical details and error handling; such things can be checked much quicker with automated system tests. Instead UAT focuses on specific human and business concerns.



Examples

What does this all mean in practice? Let's use the example of the To-Do project app. Below are a few cases you might want to test. They all relate to managing tasks, but sit at different levels of the test pyramid:

1. A user can add a new item to the to do list
2. The app can handle a response from Trello's API
3. The item sets a `completed_at` field to the current time when it is marked as `'Done'`

Test 1 is an end-to-end test. It describes a user-facing UI behaviour, without any details of how that particular task is accomplished.

Test 2 is an integration test. It tests communication between your code and an external dependency (the Trello API).

Test 3 is a unit test. It tests specific implementation details of the code in your app.

How do we write automated tests?

Lots of software has been developed to help with automated testing. The right choice depends on the level at which you're testing (i.e. unit, integration, end-to-end), and the intention of your test. We might want to automate a user clicking buttons in a browser, or an application connecting to Trello, or simply call a method on a class we have written.

Which type of test should I use?

The closer to the actual code the test is, the quicker the feedback cycle is from running the tests. For this reason, tests should be pushed down into the unit test layer wherever possible. End-to-end testing should cover as few things as possible, as it has the longest feedback cycle. Not all applications require automated tests at every level - if you can gain confidence in the application's behaviour simply by automating unit tests and integration tests, then do so!

What tools are available to help?

Every programming language has its own libraries to help you write and run code tests. Be sure to check the frameworks you're using as well for tools to help test your code! Automated testing is a standard challenge across the software industry, and there are powerful open-source solutions available.

Python has powerful libraries that will help with each type of test. Over the course of this module we'll gain hands-on experience with some of the most commonly used tools, writing unit and integration tests with pytest, and exploring UI testing with selenium.

Static Analysis Tools

Static analysis is the act of inspecting your source code without running it and asking "Does this look OK?". You can do this yourself (for example, during code review), and, just like testing, this process can be automated. Static analysis tools are an important companion to automated testing and the two are often talked about together. But they are different! Testing involves running your code, whereas static analysis simply looks at it. Static analysis serves two main purposes:

1. Detect Potential Errors
A lot of bugs can be predicted from looking carefully at source code. Static analysis tools are really good at doing this. They can give early warning of things that will fail the compilation process (for compiled languages), as well as issues that may crop up at runtime. The types of bugs flagged here are language specific. In Python you might use static analysis to check all your code is consistent with the type annotations you've used, or that all files have consistent whitespace indentation. If you happen to be a C developer, you would perhaps be more interested in flagging double frees or unhandled return codes.

2. Enforce Style
In every language there are bad practices, i.e. code structures that work but are considered inferior to alternatives. Most software teams will also have their own coding conventions and style guides to ensure code is consistent across a codebase. Static analysis tools and, in particular, '*linters*' can help by checking for, and optionally enforcing, good code style.
What sorts of things do linters check? They're usually highly configurable, but some examples include: enforcing line length limits, checking variable naming conventions, or flagging any unused code. Many such issues are not errors and therefore cannot be detected by testing. It doesn't affect how the computer runs your code. Instead, linters enforce style to make it easy for other developers to work with. This is vital to ensure the readability and maintainability of long-lived codebases.

"Programs are meant to be read by humans and only incidentally for computers to execute."
- *Donald Knuth*

Static Analysis Tooling (Python)
Static analysis tools are used across all languages and technologies. Indeed, you've probably already taken advantage of built-in static analysis as your code editor or IDE suggests an auto-completion, warns you about a name that doesn't exist or automatically handles code indentation. A few common Python static analysis tools are listed below to give you a flavour of the variety out there. This is by no means an exhaustive list!

- **Black**: code reformatter (https://pypi.org/project/black/)
- **Flake8**: a wrapper around pyflakes, plus enforcement of the (somewhat outdated) PEP 8 conventions (https://pypi.org/project/flake8/)
- **Mypy**: enforces type annotations (https://pypi.org/project/mypy/)
- **Pyflakes**: quick checking for common runtime errors (https://pypi.org/project/pyflakes/)
- **Pylint**: assorted linting options (error detection, code style) (https://pypi.org/project/pylint/)

Chapter 2
Unit Tests

In this section we'll discuss unit tests in greater depth. What are they, and how can we write them? We'll dive into Python's testing ecosystem and take a look at some of the tools available to help us. We'll also discuss test doubles and mocking strategies, and the part they play in building robust unit tests. Finally we'll dive into some general ideas about how to write good tests, and, equally important, how to write testable code.

What are unit tests?

A unit test checks the behaviour of the smallest units of your program - in Python's case, these are the functions and classes that make up your application. Unit testing sits at the base of the test pyramid – they are usually the most numerous kind of tests, and the ones that will catch most bugs. Unit tests are quick and easy to run. This rapid feedback cycle makes them the foundation of automated testing.

Imagine you're writing a messaging app which lets you send encrypted messages to your friends. The app has to do a few different things - it has a UI to let users send and receive messages, it sends the messages to other users using an HTTP API connected to a backend server, and it encrypts and decrypts the messages.

You've written the code, but you want to add tests to catch bugs - both ones that are lurking in the code now, and regressions that a developer might accidentally introduce in future. The encryption and decryption code seems like a good place to start: it's isolated from the rest of the code and it's an important component with some fiddly logic.

It uses a Caesar cipher[1] for encryption, which shifts all the letters by a given amount. It's pretty insecure, but we're pretty sure nobody will use this app to send really secret secrets...

The encryption code looks like this:

```
def encrypt(message: str, shift: int):
    # Do the encryption...
    return encrypted_message
```

The `encrypt` function takes two parameters: a message containing the string we want to encode, and a shift which is an integer from 0 to 25 specifying how many letters to shift the message. A shift of 0 means no change. A shift of 1 means shift each letter by 1, so "hello" would become "ifmmp", and so on.

We can test this function by passing in a message and a shift, and checking that it returns what we expect.

```
def test_encrypt():
    """Shifts each letter in the message"""
    message = 'abcd'
    shift = 1

    result = encrypt(message, shift)

    # assert raises an exception if the
    # following expression is not True
    assert result == 'bcde'
```

This is a **unit test**. The code inside the test has three parts:

- The *test setup*, where we specify the input to the function
- A call to the function we're testing
- An *assertion* that the function returned the value we expect

This structure is sometimes called *Arrange/Act/Assert* or *Given/When/Then*. It's a helpful structure, particularly when writing longer tests as it makes them easier to follow. This convention isn't specific to Python.

In practice, you wouldn't settle for just one unit test for the encrypt function. A good developer will think of many of edge cases, things like:

- How should it process an empty message?
- Does it work fine with a string 1,000 characters long? 100,000?
- What does it do with unicode?
- Does it handle a negative shift? Should it? If not, does it raise a sensible exception?
- What about `shift = 0`?

You could write unit tests for all of these, thinking through these questions may raise interesting design choices (for example, is it meaningful to apply a Caesar cipher to unicode text?). This is one of the main reasons developers write unit tests alongside their code - *writing unit tests forces you to think about what you're trying to accomplish, and the edge cases involved*. It results in cleaner code more closely aligned with your design goals.

Unit tests should deliberately trigger exceptions and check those exceptions are the right type. They should test edge cases so slower, higher-level tests don't need to. It's normal for a single function to have five, ten or more unit tests. One is not enough!

Unit Testing in Python

Every language has its own tools for writing and running unit tests. In the world of Python there are two main options: unittest[2] and pytest[3]. You might also hear about nose and nose2[4], although they are far less widely used, and we won't dive into them here. Testing frameworks are essential for automated testing. They provide some core features and enforce useful conventions. Core components of any testing framework are: tools to discover and run your tests (usually via the command line), conventions for writing tests, and utilities to help with common testing tasks like parametrising tests, handling expected errors, mocking dependencies and performing setup/teardown logic. Many popular testing frameworks also offer additional nice-to-have features such as plugin support and highly customisable output formatting.

Unittest

Unittest is the testing framework included in the Python standard library, and its ubiquity is a big selling point. Almost every Python installation has it available without installing any third-party packages. It's modelled on the JUnit testing framework for Java, and should feel familiar to programmers with experience in strict object-oriented languages such as C# or Java.

In unittest, test cases are written into classes that inherit from the TestCase class, and look like this example from the documentation:

```
from unittest import TestCase

class TestStringMethods(TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)
```

Unittest can then be launched by the following from Python:

```
import unittest
unittest.main()
```

You'll see unittest used widely, but the Python community has largely converged on using a third-party option - **pytest** - instead. In this course we'll use pytest to write our unit and integration tests, but you should be aware that there are other options out there. Unittest is the most common of these. Indeed, the two are not mutually exclusive - pytest supports unittest test cases natively.

Pytest

Pytest[5] is a widely used testing framework for the Python language, and we'll be using it to run tests throughout this course. Pytest isn't part of the standard library. It's an alternative to unittest. You can install pytest via pip or another package manager, for example:

```
$ pip install pytest
```

To run pytest, simply run the following command from the root of your project.

```
$ pytest
```

Here's how it works:

1. You write test functions that include assertions.
2. You run pytest.
3. Pytest automatically discovers your test functions, runs them, and reports any failed assertions.

Simple, right? The Caesar cipher example we used earlier is a valid test in pytest. Below are some further examples of tests written for pytest. The test cases we've used are identical to those shown earlier to demonstrate unittest syntax. Compare the two and note the stylistic differences between the two frameworks:

```
import pytest

def test_upper():
    assert 'foo'.upper() == 'FOO'

def test_isupper():
    assert 'FOO'.isupper()
    assert not 'Foo'.isupper()
```

```
def test_split():
    s = 'hello world'
    assert s.split() == ['hello', 'world']
    # check that s.split fails when the separator is not
a string
    with pytest.raises(TypeError):
        s.split(2)
```

You may have noticed that the syntax for writing pytest is much simpler than unittest. There are no classes required, and no inheritance from the testing library. Simply name a test function appropriately, use a native python assertion, and you're good to go. This ease of use is perhaps the greatest strength of pytest, and it has several other more technical advantages over unittest. We'll discuss some of these, for example parametrisation and fixtures, in this module. Other advantages that we won't dive deeply into include:

- A rich plugin ecosystem
- Highly configurable test runner
- More informative error reporting

Discovery

We've mentioned that pytest can automatically discover your tests. This is a great feature, but how does it work? As a test-writer, you need to ensure you've named your directories, files and functions in ways that pytest will understand. Pytest is highly configurable, but there are a few default rules that you're unlikely to change:

- Test function or method names must be prefixed with "test".
- If used, test class names must be prefixed with "Test".
- Tests must live in files named `test_*.py` or `*_test.py`

A common test directory structure is shown below. It's common to keep unit tests in a separate directory from your test code, although this is not a requirement and conventions will vary between software projects. Inside the test package you might create any arbitrary tree of sub-packages to organise your test files. It's common to use a test package structure that mirrors the package structure in source code, but this is not required.

```
src/
    mypkg/
        __init__.py
        ...
tests/
    __init__.py
    foo/
        __init__.py
        test_app.py
```

And the code within `test_app.py` might look something like:

```
from mypkg.foo import App

def test_app():
    app = App()
    app.run()
    assert app.is_running()
```

There are many other ways to structure your tests, and you can read recommendations and tips on the pytest documentation[6]. You can even change pytest's discovery behaviour entirely to suit your needs (see Pytest API Reference[7]), although we recommend only using this as a last resort.

Fixtures

All tests require some set up; you need to create variables and objects to test, and get them into the right state to be tested. This kind of setup can get very repetitive, and you'll often end up with multiple tests with the same lines of "Arrange" code:

```
def test_car_moves():
    car = Car()
    car.add_driver()
    car.start()
    ...

def test_car_stops():
    car = Car()
    car.add_driver()
    car.start()
    ...
```

These two tests both do the same setup steps - in this example there are three lines of code to set up a car object. In more complex situations common setup and configuration could take tens or even hundreds of lines of code! Code repetition is usually a bad thing – it's difficult to read and maintain. Software developers often talk about DRY (Don't Repeat Yourself) when writing application code, and the same principle applies to test code. Remember, you test code should be as clean and well-maintained as your application code.

Testing frameworks come with tools to help you commonise this kind of setup code, and also perform any necessary cleanup or teardown steps. In some testing frameworks these are called "SetUp" and "TearDown" methods. Pytest approaches the problem slightly differently, with elegant solutions known as "**fixtures**". The code above can be re-written to use a fixture:

```
import pytest

@pytest.fixture
def moving_car():
    car = Car()
    car.add_driver()
    car.start()
    return car

def test_car_moves(moving_car):
    ...

def test_car_stops(moving_car):
    ...
```

So, what's going on here? We have moved the common setup code into a separate function and annotated it with the `pytest.fixture` decorator. The result is that the tests can take a parameter called "`moving_car`" and pytest knows to provide this parameter with a value returned from the `moving_car()` function. The fixture is called once per test, so each test gets a separate Car object. Tests can then focus to test-specific details (the "Act" and "Assert" parts), while sharing the common setup logic.

Fixtures can also contain teardown code. Imagine you're using a fixture to provide a HTTP client to some test code, and need to make sure the connection is properly closed after every test. What will you do? You don't want to add another line to the end of every single test. Instead you can use the `yield` keyword instead of `return` – all code after `yield` is executed after the test, and can contain teardown instructions. Teardown instructions are commonly used to ensure file, database and http connections are closed correctly, although there are many more creative uses!

Pytest also comes with some built-in fixtures to help with common test tasks. For example, the **monkeypatch** fixture provides tools to help with mocking functionality (more on that later), and the **tmp_path** fixture provides a test-specific temporary directory to help with tests that require disk IO. You can read about these, and other, built-in fixtures in the documentation[8].

```
@pytest.fixture
def http_connection():
    http_connection = http.client.HTTPConnection('www.python.org')
    yield http_connection  # provide the fixture value
    print("teardown http client")
    http_connection.close()
```

Handling Errors

When writing tests, you often want to check that a particular kind of error or exception was raised. This may sound odd, but a key part of testing is checking that error handling works as designed, and the first step of that is ensuring that the correct `Exception` types are raised in the right places.

Without any special handling, pytest (and other frameworks) will fail if an exception is thrown while running a test. An exception will be treated much like a failed tested case. To tell pytest that an error is expected, you can use the `pytest.raises()` context manager:

Sharing Fixtures

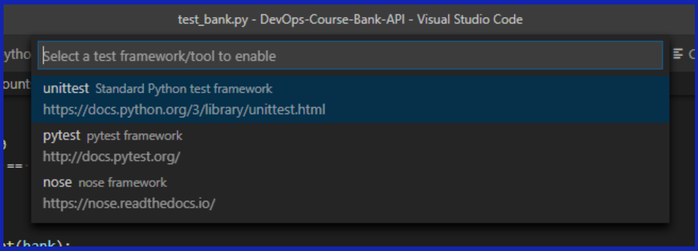
You can share fixtures between multiple test files by defining them in a `conftest.py` file.

```
1    def test_cant_add_number_to_string():
2        text = 'a'
3        number = 3
4        with pytest.raises(TypeError):
5            text + number
```

The test above will only succeed if line 5 throws a `TypeError` (or subclass). If line 5 runs successfully, or an exception is raised elsewhere, or the exception is not a `TypeError`, the test

Debugging Unit Tests in VS Code

Setting up debugging for unit tests in Python is thankfully quite straightforward. You can instruct VSCode to discover these automatically by selecting `Ctrl+Shift+P => Discover Tests` upon which VSCode will ask what testing framework you are using (unitest, pytest, etc.):

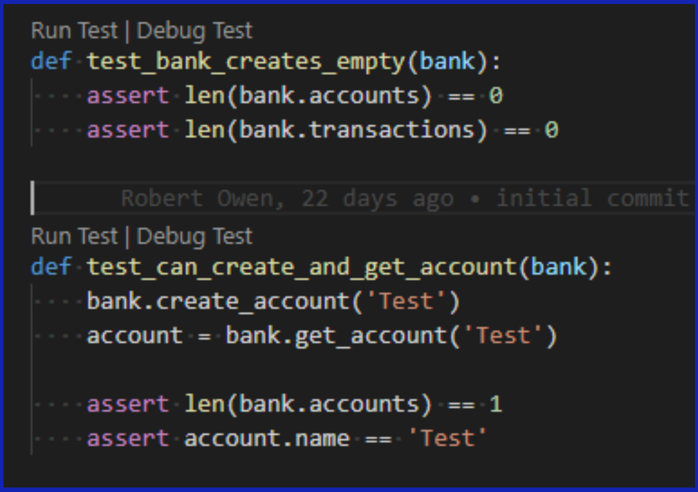


test_bank.py - DevOps-Course-Bank-API - Visual Studio Code

```
Select a test framework/tool to enable
unittest  Standard Python test framework
https://docs.python.org/3/library/unittest.html
pytest  pytest framework
http://docs.pytest.org/
nose  nose framework
https://nose.readthedocs.io/
```

Select Testing Framework Image

After doing this CodeLens annotations will appear above discovered tests:



```
Run Test | Debug Test
def test_bank_creates_empty(bank):
    assert len(bank.accounts) == 0
    assert len(bank.transactions) == 0

        Robert Owen, 22 days ago • initial commit
Run Test | Debug Test
def test_can_create_and_get_account(bank):
    bank.create_account('Test')
    account = bank.get_account('Test')

    assert len(bank.accounts) == 1
    assert account.name == 'Test'
```

CodeLens Test Adornments

From here you can set breakpoints and start debugging as you would normally.

Using Poetry with VSCode
In order to get VSCode working nicely with poetry you will need to install the virtual env folder in a place where VSCode can find it.
An easy way to do this is to change the poetry config to store the venv folder in your project folder:

```
poetry config virtualenvs.in-project true # to set this globally
poetry config --local virtualenvs.in-project true # to set for the current project only
```

Next you'll need to re-run poetry install to actually trigger the generation of the .venv folder.

After this VSCode should notice the virtual environment folder and ask if you'd like to use it (say yes). This sets the "python.pythonPath" value in `.vscode/settings.json`. From here you can run `Ctrl+Shift+P => Discover Tests` as described above.

will fail. This specificity is extremely useful for ensuring that the specific errors you expect are created, and not any others.

If you want to allow any error, you can use the base `Exception` class in the context manager instead:

```
with pytest.raises(Exception):
    ...
```

However, this is bad practice. You should be as specific as possible with the errors you allow. Allowing all `Exceptions` raises a very real risk of allowing bugs through your test suite.

Parametrised Tests

You'll often want to run a test multiple times with slightly different parameters. Using the techniques described above this would be tricky - you'd need to have multiple copies of the test code, one with each parameter! Clearly there must be a better way to do this. It's called a **parametrised test**, and pytest provides this functionality through a built-in decorator. The example below shows a test that will be run *four times*, with the parameters 0, 1, "hello" and "world" in turn.

```
@pytest.mark.parametrize("value",
[0, 1, "hello", "world"])
def test_process_value(value):
    …
```

Handling Multiple Parameters in Unit Tests

If you want to parameterise multiple parameters in a unit test you can use tuple syntax (similar to how you can return multiple values from a function):

```
import pytest

@pytest.mark.parametrize("test_input, expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

It's worth mentioning that when dealing with a large number of test parameters it's often cleaner to assign the test parameter collection to a variable beforehand:

```
import pytest

test_params = [("3+5", 8), ("2+4", 6), ("6*9", 42)]

@pytest.mark.parametrize("test_input, expected", test_params)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

This also allows for the params to be shared between tests.

Code Coverage

How do you know your tests are good enough? This is a really tricky question to answer, and depends on many factors, but a key metric is **code coverage**. Code coverage tools measure how many lines of code are run by your tests. While this is a limited and over-simplistic way of measuring test suite quality, it is a very useful approach for highlighting areas where you need more tests.

Software projects will often have testing requirements expressed in terms code coverage (e.g. 90%). Such metrics can be useful but treating them as a goal can be dangerous - such practices encourage writing useless tests simply to increase the coverage score. Make sure every test you write adds value, regardless of code coverage.

Pytest has a plugin, pytest-cov[9], that provides code coverage reports for python projects. Once installed, you can optionally include coverage reporting when you invoke pytest by running:

```
pytest --cov=<PATH_TO_PACKAGE>
```

Alternatively, to always generate coverage data when running pytest you can add a `pytest.ini` file in your project root and specify the coverage option there:

```
[pytest]
addopts = —cov=<PATH_TO_PACKAGE>
```

A Warning: Code Coverage Metrics

Don't confuse code coverage with test quality. Code can contain plenty of bugs, even with 100% code coverage. Code contains logic, and good tests need to check this logic for edge cases, special values and configuration adjustments. Running a line of code once is not enough.

It's very easy to write bad tests that achieve good code coverage numbers. Code coverage should always be evaluated alongside more nuanced metrics (e.g. branch coverage), and with confidence that test quality is consistently high through other procedures (e.g. code review).

Writing Good Tests

Unit tests are a form of **White Box testing**. This means the tester has full knowledge of the internal workings of the code under test, and should use that knowledge to try to find edge cases and failure conditions. The tester will write unit tests to check those edge cases, as well as the "happy cases" where the code runs with typical inputs. Being thorough at this stage both ensures the code works as intended, and provides safety-checks for many possible bugs that could be introduced in future as the codebase changes.

There's no fixed rule on how to approach writing unit tests, but there are some common themes that will help guide you:

- Keep them short
- Write a lot of them
- Avoid tight coupling to implementation details
- Test your requirements
- Test edge cases

In fact, writing unit tests should be easy. Having trouble? Take a look at the code you're testing - the problem is often with the code, rather than your tests.

Writing Testable Code

Code that follows SOLID principles should be very easy to test. Unit testing can become very difficult, if not impossible, if these principles are not followed closely. This isn't always obvious. It's very easy to write functional, bug-free, readable code that is very difficult to test.

Sometimes a developer may ask "If the code is readable and works, what is the problem? Tests exist to check my code, not the other way around!" The problem is that, although the original developer might know it works, nobody else does. If the code can't be unit tested, it can't be safely edited later, and other members of the team need to read through the source code to be sure it works as advertised. That might work for a very small codebase, but in a large codebase with hundreds of classes it's not reasonable to expect developers to know all the source code.

Let's start with an example of a readable, but hard-to-test function:

```
from datetime import datetime

def is_morning():
    hour = datetime.now().hour
    return hour < 12
```

What's wrong here? Well, this function violates the single responsibility principle. It does two things: it gets the current time and determines whether that time is in the morning. The function is also non-deterministic: calling it twice will give different answers depending on the system clock. This makes it particularly difficult to test, as we can't directly control the "hidden input" (`datetime.now`).

How would you test this? You could try something like the snippet below, however *don't do this* (for reasons discussed below):

```
1    # Bad test
2    def test_is_morning_am():
3        # Arrange
4        now = datetime.now().hour
5        expected_answer = now < 12
6        # Act
7        result = is_morning()
8        # Assert
9        assert expected_answer == result
```

This is a bad test for several reasons. Firstly, the test re-implements the functionality of the code under test to build an expected answer. This tightly couples the test to the code's implementation details, and it will be very brittle if the code is changed in future. Secondly, the test is not the same each time the code is run! Automated tests need to be repeatable, and having them coupled to an external input (e.g. the system clock) makes this extremely difficult.

Finally, the test may fail because there is the times generated on lines 4 and 7 are not identical. They will probably differ by a couple of nanoseconds, but it could be more for a variety of reasons. Run this test

Automatic Test Generation with Hypothesis

If you have to generate a lot of unit tests, there are tools out there to help. For Python, a popular library is **hypothesis**: https://hypothesis.readthedocs.io/en/latest/.

Hypothesis automatically generates parameters for your tests – lots of parameters – and reports back the simplest possible version of each failure case. This can save you lots of effort manually parametrising tests, as well as finding edge cases that you simply wouldn't think of. Hypothesis can provide you with any kind of input data you can think of, including strings, numbers, lists as well as more complex objects generated from randomised inputs. You can fine-tune the range of possible value, filter out certain exceptional inputs you don't want to test for, and much more.

Try the code below for an example. A division test is run twice: first it's parametrised using pytest's built-in functionality to run it five times with the specified integer inputs. The second time we ask hypothesis to provide input values. Libraries like hypothesis will save you time writing test code and result in more readable, flexible and robust test suites.

```
import pytest
from hypothesis import given
from hypothesis.strategies import integers

@pytest.mark.parametrize('number', [-10, 0,
1, 5, 1000000])
def test_division(number):
    assert number / 1 == number

@given(number=integers())
def test_division_with_hypothesis(number):
    assert number / 1 == number
```

Run the code. The first test passes for all five values, but the second hypothesis-based test fails! It's not a case most developers would immediately think of, and that's precisely why something like hypothesis is valuable. Think about why test fails for extremely large values, although don't worry if the answer isn't obvious.

at 11:59:59.999999 AM and you may see a failure, as the `is_morning` function runs in the afternoon whereas the *Arrange* statements ran in the morning. Such errors are very difficult to reproduce and debug.

Instead, if at all possible you should refactor this code to adhere to the single responsibility principle. We've discussed refactoring strategies before and will come back to the topic again. In this case the code is very simple, and we can jump to the end point below. We've refactored `is_morning()` to take a `datetime` object as an argument, enabling us to write some much better unit tests:

```
from datetime import datetime
import pytest

def is_morning(dtime):
    return dtime.hour < 12

# 0:00 to 11:00
@pytest.mark.parametrize("hour", range(12))
def test_is_morning(hour):
    dtime = datetime(year=2020, month=1, day=1, hour=hour)
    assert is_morning(dtime)

# 12:00 to 23:00
@pytest.mark.parametrize("hour", range(12, 24))
def test_is_not_morning(hour):
    dtime = datetime(year=2020, month=1, day=1, hour=hour)
    assert is_morning(dtime) is False
```

With this refactoring, it's easy to write unit tests that check every hour of the day for the correct result, without replicating the implementation details of the tested function. The unit tests are now simpler, more comprehensive and repeatable.

SOLID principles, introduced earlier, are the key to writing testable code. Make sure you understand them and know how to use that knowledge in your day-to-day programming. Keep in your mind the question "How would I test this?", and you should identify any code design issues early.

Test doubles and mocking strategies

Unit tests run little pieces of your application code in isolation. As we've discovered, writing good, testable code supports this by providing clean boundaries along which you can slice your application into functional pieces. But you can't always cleanly separate every functional unit from the others. Sometimes the system under test needs secondary objects to function. This presents a challenge: how do you write unit tests that test only a single class, while also including all the other classes needed to correctly configure the class being tested? This challenge may not be apparent in small codebases, but it's a common hurdle in production codebases in all object-oriented languages (including Python). High level objects will typically require lots of other objects passed into their constructors.

One option is to use a real instance of your dependency. This will couple your unit test to another function or class, and all its dependencies so take care! It can make it harder to diagnose the source of a test failure, but you can be sure that your unit test is running code in a production-like way.

Sometimes you don't want to use real dependencies. Perhaps your testing style requires that the unit test covers one and only one class, or perhaps the real dependency has side-effects you need to avoid for testing purposes. In both cases the answer is to use a **test double**. A test double is an object used in place of a real object for testing purposes. The aim of the tests isn't to test the double (that would be silly), but to use the double to facilitate testing another object that we're really interested in. Test doubles are part of the essential toolkit for writing unit tests in object oriented languages. They are also useful in integration tests.

Mocking Strategies

"Test double" is a broad term that encompasses a few different strategies and goals. In all cases, the purpose of a test double is to limit the scope of a test to the system/class/function you're interested in. The particular kind of test double used will depend on what you need to achieve to satisfy the test logic. There are three main types:

A **Stub** object has no logic, and returns pre-programmed values to calls on its methods. Stubs are extremely important, and are commonly used when writing unit tests.

A **Spy** object records details about how it was used (e.g. what methods were called, how many times, and with what parameters). Frequently a spy is also a stub. The details recorded by a spy can be important to validating test success, particularly if the code under test has side-effects. Spies can be used to check that the side-effects were triggered correctly.

A **Fake** object is more sophisticated than a stub. It has real functionality, but take some shortcuts to facilitate testing. A good example would be using a in-memory database, or dropping a slow validation step used in production code. Fakes are not very useful for unit testing, as they still contain logic. Remember, unit testing is about running logic only in the class or function under test. However, fakes are extremely important when talking about integration testing, a topic we'll come back to later.

You should be aware of these three types of test doubles, and when you might use each. In practice, the divisions are often not clear cut (you might stub one method, and fake another). Test doubles of all categories are often simply known as "*mocks*", and the process of creating them from real objects is called "*mocking*".

These mocking strategies are not Python-specific – they are used across all object-oriented languages. Python is very amenable to mocking objects due to its open nature. Testing tools can freely modify dependency code to suit your needs. You'll find that creating test doubles can be significantly more complex in compiled languages with privacy models like Java and C#, although the core concepts are the same and these languages have their own tools to help.

Python Tools

In Python there are two main libraries that are used to create test doubles.

Firstly, unittest.mock[10] is bundled in the standard library. Despite its name (a sub-package of unittest), it is not tightly coupled to the unittest framework and can be used easily with other testing frameworks (e.g. pytest). unittest.mock is an extremely powerful and versatile library that can create Stubs, Spies and Mocks quickly and safely. We won't dive deeply into this library here, but you should be aware that it exists!

Secondly, pytest provides some mocking functionality through its monkeypatch[11] fixture. You don't need to configure anything extra to use this, simply add the '`monkeypatch`' argument to a test and pytest will inject the fixture. Monkeypatch has helpful methods for replacing object attributes, environment variables, items and adjusting the current directory. Changes made by monkeypatch exist for the duration of the current test, and are automatically reverted afterwards.

An Example

Let's take a look at an example. Consider the following Python code for shearing virtual sheep:

```
from random import random
from typing import Iterable

class Sheep:
    def __init__(self, weight: float):
        self._weight = weight

    def get_wool(self):
        return random() * self._weight


class Shears:
    def __init__(self, efficiency: float):
        self._efficiency = efficiency

    def shear(self, targets: Iterable[Sheep]):
        wool = 0
        print('Sharpening shears...')
        for target in targets:
            print(f'Shearing a {target.__class__.__name__}')
            wool += target.get_wool()

        return wool * self._efficiency
```

Home-made mocking

In Python, you have access to all the code used by your dependencies. The language has no concept of "private" code like you will see in many compiled languages like Java or C#. This makes mocking objects and functions much easier than in less flexible languages. It may be tempting to avoid mocking libraries entirely and do it all yourself. For example, you could simply change standard library functions:

```
import platform
print(platform.python_version())
# prints: 3.8.1

platform.python_version = lambda: 'ECMAScript
2016'
print(platform.python_version())
# prints: ECMAScript 2016
```

However, there are still good reasons to use libraries like unittest.mock or pytest. Most importantly, mocking libraries provide guarantees about when they will revert your changes. There is no reason to take this on yourself - it can be tricky to implement properly, and failure to revert changes to shared libraries might have unpleasant consequences. In addition, mocking libraries will give you helper functions that will save you time and effort! Don't re-invent the wheel.

In this code we have a `Sheep` class, and a `Shears` class. Sheep produce wool in random quantities between zero and their weight. Shears can gather wool from a collection of sheep, but suffer a loss determined by their efficiency. Below is an example of using these classes together, and some example output.

```
flock = (
    Sheep(10),
    Sheep(4.3),
    Sheep(2)
)

shears = Shears(0.8)
wool = shears.shear(flock)

> Sharpening shears...
> Shearing a Sheep
> Shearing a Sheep
> Shearing a Sheep
> Wool: 7.133562646044543
```

How would you go about writing unit tests for these classes? Let's start with the `Sheep` class. Its key requirement is that `get_wool` should always return a number between 0 and the sheep's weight, as specified in the constructor. There's some randomness here. You could run a test 1,000 times and check the number is always in bounds. That'll do for now.

But what about the `Shears` class? The `shear` method expects an `Iterable` (e.g. a `List`) of `Sheep`. It shears each of them, generates a total amount of wool and reduces this according to an efficiency factor. You can't just create some `Sheep` and shear them to test it, because that would involve running logic from the `Sheep` class as part of your `Shears` unit tests! Doing that could mean that your `Shears` unit tests break in future when someone changes the `Sheep`'s wool generation algorithm. Not a good idea.

Instead you need to create something that looks like a sheep, but replaces the `get_wool` method so we aren't running the actual implementation. To be precise, we want to stub the `Sheep.get_wool` method. Let's talk about two ways to do this. Firstly, we could subclass `Sheep` to create a mock, then write a test using `MockedSheep` in place of `Sheep`:

```
class MockedSheep(Sheep):
    def get_wool(self):
        """This is a stub"""
        return 1
```

Alternatively, we could use pytest's monkeypatch[12] fixture to replace `Sheep.get_wool` class with a stubbed version. This technique is a bit cleaner, and we don't need to introduce a new class. Monkeypatch takes care of reverting out change after the test has completed.

```
def test_shears(monkeypatch):
    # Arrange
    wool_per_sheep = 1
    efficiency = 0.5
    monkeypatch.setattr(Sheep, 'get_wool', lambda self: wool_per_sheep)
    flock = (
        Sheep(5.4), # weight is irrelevant
        Sheep(2.1)
    )
    shears = Shears(efficiency)

    # Act
    wool = shears.shear(flock)

    # Assert
    assert wool == len(flock) * wool_per_sheep * efficiency
```

Using monkeypatch, we can quickly stub the functionality of the `Sheep` class. This lets us write unit tests for `Shears` that don't run any logic on the `Sheep` class.

Monkeypatch isn't limited to changing attributes on your own classes. You can also override behaviour on imported code from the standard library or third-party packages. You could, for example, write better unit tests for `Sheep` by stubbing out the `random.random()` method used to randomise each sheep's wool output.

Chapter 3

Test Driven Development

In this section we'll introduce Test-Driven-Development (TDD). We'll briefly cover the practice's philosophy and goals, as well as look at practical tips on how to apply TDD in your everyday work. We'll discuss the advantages and disadvantages of TDD, and see how the workflow can be combined with pair programming.

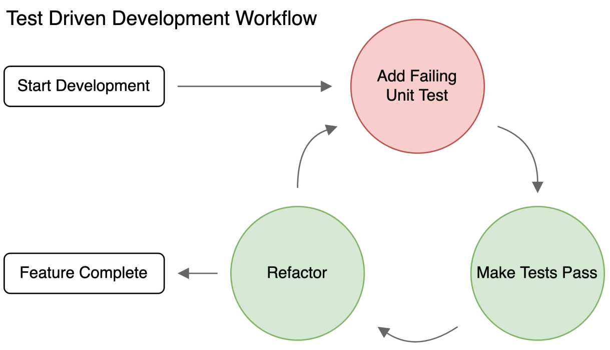
What is Test Driven Development?

TDD is a software development methodology where developers write tests before they write code to satisfy those tests. This may sound counter-intuitive and a bit odd, but stay with us.

In traditional software development, developers write application code, and later someone (perhaps a different person) writes unit tests to ensure that code is operating correctly. The application code is written to satisfy requirements, then tests are written to check that functionality. This works, but for decades people have struggled with buggy and unreliable software. Developers have spent a lot of time thinking "Is there a better way to do this?" TDD is one answer to this question, and it specifically attempts to address some key issues with this traditional approach described here:

1. Tests may never get written (resources re-prioritised)
2. Tests may be incomplete
3. Tests check functionality, not necessarily requirements

When practicing TDD, developers work on a strict, short feedback cycle to implement features. New tests cases are written based on requirements, and will initially fail. The classes they try to use may not even exist! Once a test is written you can go ahead and write code to satisfy the test and *nothing more*. Once the tests are passing, do any necessary refactoring and start the process again.



This cycle is sometimes known as **Red-Green-Refactor**, where red and green refer to the state of your unit tests at that stage of the cycle (i.e. red = failing). There are a few key points:

1. **Add one test at a time**. Limiting to a single test keeps the cycle quick, which keeps development focused. Make sure the new test fails. If it doesn't you have either duplicated an existing term, or have previously introduced untested code.
2. **Add only the code needed to make your test pass**. If you add more functionality than the test covers, then that functionality is untested! Write only what is needed to make your tests pass.
3. **Refactor often**. The third step in the cycle is to refactor. You don't need to do this on every cycle (there's often nothing to do), but it's included to remind you to continually consider refactor opportunities. Keep refactoring small and often.

You Ain't Gonna Need It (YAGNI) is a commonly-used acronym in software engineering. The idea is: don't write unnecessary code! Only add features when you have need of them. This helps keep codebases tidy, and delivers MVP functionality quicker. YAGNI is particularly relevant when following TDD. Every one iteration through the Red-Green-Refactor loop is an exercise in YAGNI - write the simplest code that satisfies your test.

Strict vs Pragmatic

There's always a need for common sense when writing "only what is needed to make your test pass". For example, imagine we want to write our own code to check if a number is even, following TDD. We'd start by writing a unit test:

```
@pytest.mark.parametrize("number",
[0, 2, 4, 100])
def test_is_even(number):
    result = is_even(number)
    assert result is True
```

What's the simplest code that will make this pass? Code doesn't get much simpler than this:

```
def is_even(number):
    return True
```

Obviously this is not what was intended, but it is correct functionality from a TDD point of view. Your tests now pass. From here, you decide to add a second unit test to check some odd numbers:

```
@pytest.mark.parametrize("number",
[1, 3, 5, 101])
def test_is_odd(number):
    result = is_even(number)
    assert result is False
```

What's the simplest code that would now work? This is where things can get a bit silly. Suppose you update the is_even function to this:

```
def is_even(number):
    return number in [0, 2, 4, 100]
```

The solution is arguably simpler than a "correct" implementation, but we're now in a position where the tests look reasonably thorough, but obviously the code is wrong. Writing code during TDD is a balance between naively satisfying the tests, and avoiding simply hard-coding the test specifics into your application code.

A particularly firm believer in TDD might disagree here and say "just keep adding unit tests and eventually the simplest solution will be the correct one". This is certainly true. There are tools that can help you randomise test inputs, making this kind of hard-coded response impossible (e.g. hypothesis). However, writing huge numbers of unit tests to cover trivial extensions of existing test cases is generally not the best use of development resources and degrades test suite performance. A pragmatic middle-ground needs to be found on each project, and enforced by pair programming and/or code review.

Advantages

Adopting TDD has numerous advantages.

1. **Fewer bugs**. TDD's emphasis on writing unit tests usually results in high code coverage, testable code and high-quality unit tests. This results in fewer bugs making their way the finished product. While the product may take longer to get there (see disadvantages), the quality is likely to be higher.
2. **Closer alignment to specifications**. Writing tests before features forces you to think carefully about the functionality needed, and will usually result in fewer unnecessary lines of code (YAGNI).
3. **Self-testing code**. TDD forces you to build up a suite of unit tests of a quality unlikely to arise through more traditional workflows. A good test suite is a huge aid to any developers maintaining the product in future. The focus on testability also keeps the codebase modular.
4. **Self-documenting code**. Well-written tests provide clear documentation and, unlike external documentation, they cannot become out of date.
5. **Fewer broken tests**. It's surprisingly easy to write a unit test that never fails! The TDD cycle ensures you start from a failing state. This makes it hard to write a completely broken test.

Disadvantages

It's not all positives though. There are some key downsides to following TDD.

1. **Slower MVP delivery**. Test driven development can delay getting a minimum viable product (MVP) to market, compared to a more relaxed testing approach. Instead, TDD focuses on pre-empting possible failures and decreasing long-term maintenance burden. If time-to-market is your number one priority, you may wish to adopt a more feature-driven approach to software development, applying tests more selectively to key functionality.
2. **Not as fun**. It may sound trivial, but TDD forces a methodical turn-the-handle approach to software development, whereas otherwise it can be a more creative pursuit. TDD reduces opportunities to view an architecture and come up with a single, brilliant solution in code. Many software developers find this less enjoyable, even if the output is higher quality.
3. **Low value tests**. Strict TDD requires that you write unit tests for everything. This means writing a lot of tests for trivial functionality, or spending a lot of time creating complicated mocks just so you can write the unit test in the first place.
4. **Tight Coupling**. There's a risk that excessive unit testing creates tests tightly coupled to implementation details. This can be particularly apparent when testing methods that delegate most work to dependencies. In situations like this you can end up testing the implementation details of mocked dependencies, which is not what you wanted!

It's important to keep in mind that pure TDD can be rather proscriptive and inflexible. Instead we'd recommend a pragmatic approach taking TDD and its lessons for some aspects of your work, but also recognising areas where it may not be helpful. One of the risks of TDD is a tendency to over-emphasise unit tests at the expense of integration testing, and occasionally stepping out of the TDD mindset can help catch this.

Remember, the goal of automated testing is to give you confidence that, if the tests pass, your software is ready to release. To reach this point you'll need lots of tests, but in the end it is not important how they were written. What's important is that the tests are good enough and numerous enough to provide this assurance. Strict TDD is one methodology to reach this goal and it works very well in some circumstances, but it's not something to pursue for its own sake.

TDD and Pair Programming

TDD and pair programming are a natural fit. In TDD there are two distinct roles, which neatly map to the driver & navigator roles in pair programming:

1. Tester. Writes failing unit tests.
2. Programmer. Writes code to satisfy the unit tests.

If you're following TDD alone you alternate between roles while moving through the Red-Green-Refactor loop. Naturally, if you're working in a pair you can take on a role each. One person becomes the tester, and the other becomes the programmer. This is similar to the traditional driver-navigator split during pair programming, but gets both members of the pair actively involved in writing code. The driver writes test cases, then the navigator implements code to satisfy those test cases. Switch roles each time!

- Person A writes a test
- Person B makes it pass
- (Refactor)
- Person B writes a test
- Person A makes it pass
- ...

This style of pair programming is sometimes known as "Ping-Pong Pairing", presumably because the keyboard switches hands very frequently!

Chapter 4

Integration Tests

We've already introduced integration tests. They sit between unit tests and E2E tests in the testing pyramid. Integration tests check our system components work together. In this section we'll talk about a little more about the practical details of integration testing. What should be integration tested, what shouldn't, and how do we write these tests in Python?

What To Test?

Integration tests check that the interfaces between system components work. This may sound a little vague, and it is! That's because integration tests can span a wide range of complexities and levels of abstraction depending on the software architecture. For example, the following cases could all be considered integration tests:

- Ensuring two classes operate well together.
- Testing your CLI tool can connect to a server.
- Testing three separate microservices are able to communicate effectively.

An integration test checks code components work together, and integrate properly with third-party tools and libraries that may also form part of your application. It should not re-test functionality already checked by unit tests. Instead, it typically focuses on the interfaces that were mocked in unit tests. Similarly, integration tests do not need to run on production-like infrastructure, and can largely disregard non-functional requirements such as performance and security. Those are concerns for system tests.

Integration tests will typically use far fewer mocks than unit tests, although they are still used. In particular, you may find it useful to fake a slow database or external API during integration testing to keep test performance high, and avoid any unnecessary charges for using paid external services.

In most environments integration tests can be run on a local development machine. They, like unit tests, give rapid feedback to developers and catch many potential issues before they are even committed to the source control repository. It's normal to run integration tests after unit tests, as there's little value in finding integration errors if the unit tests are failing.

Writing Integration Tests

Integration tests are varied, and so are the ways in which you write and run them.

Simple Integration Tests

Tests that check component interaction internal to a single application or service can look at lot like unit tests! You'll write them using the same tools as unit tests (e.g. pytest[13]). They're often quick to run on a development machine. While these kinds of integration tests are superficially similar to unit tests in tooling and operation, they have different goals and the code will typically feature a lot more setup in the Arrange step, and far less mocking, than unit tests. You'll see, and write, tests of this kind as part of this modules workshop and exercise components.

Integration tests for single web services will frequently involve launching a development server to test a web application. Most web frameworks come with a development server for just this purpose, which can usually be launched with near-zero configuration for integration testing and local development. For your ongoing project, it's helpful to know that **werkzeug** (the framework Flask is built on) provides a development web client that is extremely helpful for integration testing:

```
import pytest
from myapp import app


@pytest.fixture
def test_client():
    """This fixture provides a flask test client"""
    with app.test_client() as client:
       yield client


def test_get_root(test_client):
    """Check a GET request to root path works"""
    response = test_client.get('/')
    assert response.status_code == 200
    # additional response checks go here
```

Multi-system Integration Tests

Sometimes integration tests aren't quite that simple! Integration tests often need multiple services, some of them external, to talk to each other in ways that are tricky, or impossible, to simulate within a pytest fixture (or similar setup and teardown functionality in other testing frameworks). You'll still use a test framework like pytest to write and run your tests, but you'll need to rely on external setup for dependencies and third-party services.

There are loads of possible external dependencies in software projects. Here are just a few:

- Databases
- OAuth providers
- Rate limited APIs
- Logging services
- Live applications holding critical data
- Many more!

To plan an integration test suite you'll need to decide how to handle each of these. The details will be specific to your particular project, and the services you need to talk to. Nevertheless there are a few common questions to ask yourself:

1. **Can I mock it? If so, when *will* we test this dependency?** Sometimes it won't be feasible to communicate with a dependency at all during testing. For example, you may have a limited number of licenses for a service or the cost of using a paid service is too great for routine testing.
2. **Do any of the external services we use provide tools for testing?** Some web services will grant free accounts for test purposes, or provide validation-only functionality that can help make your tests safe.
3. **Can we run integration tests locally? Or do we need to launch them from a server?** It's a great help to be able to run integration tests from a development machine - tests can be run faster and more frequently. However, you might need to run tests on a server if the build process is extremely slow, not compatible with development operating systems, or perhaps you need a DNS name to test your OAuth flow. We'll come back to this topic when discussing CI pipelines.

As well as talking to third party services, your integration tests may need to test interactions between many of your own services. Often this means checking out code from multiple source controlled repositories, deploying them all and running each repository's integration tests! That's a bit more complicated than executing `pytest`, and typically requires a more complicated script or pipeline to ensure you can run the tests reproducibly.

In later modules we'll come back to the complexities of setting up for multi-system integration tests when introducing immutable infrastructure and containers. We'll learn how to use modern containerised tools to set up copies of production infrastructure locally, and how this can help us set up and tear down live and test setups with ease.

Integration Testing with Databases

Databases are perhaps the most common external dependency in software testing. Integration tests should not usually communicate with 'real' production databases. Doing so risks contaminating live data, and will put unnecessary load on production infrastructure. However, it's still important to check that your app can work with a database! To test against a database you need to be able to connect to it, guarantee an initial state (i.e. test data) and clean or delete the database when the tests complete. Ideally test setup should support multiple test suites running in parallel, for example on many development machines at once.

For all these reasons database integration tests are rarely served well using a single persistent database hosted remotely. Instead, there are several common patterns used to run frequent integration tests against local, often temporary, databases. Some of these are described below. We'll come back to these topics when we introduce database technologies properly, so don't worry about the details at this stage.

- **Persistent local databases on development machines**. Each developer runs a local DB server for testing purposes. This works, but can require a lot of work to set up a new machine - it's not very transportable. There's also a risk of configuration drift over time.
- **Temporary local file-based or in-memory database**. Rather than running a typical database server, some databases are based on local files or can be run entirely in memory. For example SQLite and H2. These can be extremely convenient, but are limited as they're not identical to the databases you'll typically be running in production environments. This can hide real errors and create spurious errors in other places!
- **Container-based databases**. Temporary container-based databases created using immutable infrastructure tools such as docker can provide the best of both worlds: they can run the same database servers you might find in production, while also mimicking the disposability of in-memory or file-based databases. We'll talk about immutable infrastructure tools throughout this course. For Python, there are convenient libraries such as testcontainers[16] that can help with this setup.

If temporary databases are unsuitable for your use case, a persistent set of server-hosted databases are the best option! Such databases should be used solely for integration testing, and you'll need to ensure the databases are cleaned after each test run. Database configuration should be kept up-to-date with that used in production.

Chapter 5

End to End Tests

Here we'll dive a little deeper into E2E testing, and flag some key points to consider when writing any kind of E2E test. Keep in mind that this kind of testing encompasses both automated system tests and manual user acceptance tests. Our focus here is on automated tests.

What To Test?

E2E tests sit at the top of the test pyramid. They won't be very numerous, and won't be run nearly as frequently as unit or integration tests. Nevertheless, they are an essential step for validating that your app can deploy and run successfully in the production environment, and they provide a final place where you can test any slow external integrations that are unsuitable for routine integration testing.

Automated system tests should:

- Test whole-app functionality
- Test non-functional requirements
- Run on infrastructure that is similar to the production environment (where possible)

They should not:

- Test things that can be tested via unit tests or integration tests
- Contain any mocks

UI Testing

In E2E testing, we want to test our code in a way that is as similar to the end user as possible. Our tests need to interact with the codebase just like a user would. That means different things for different projects. Users might interact with your code via a native desktop application, mobile app or a web interface. All of these interactions might be modified by the use of accessibility tools, for example screen-readers. Alternatively maybe you're developing a technical product, and the end user is a GraphQL or JSON API client, or another project that embeds some library code you've published.

We won't explore approaches to all automating all of these test cases (although you should know all are possible!), but instead will focus on the most common - how can we automate tests that interact with a website?

Selenium

Selenium[17] is a popular open-source UI testing library. It creates language-specific bindings for programmatically controlling a web browser so you can write tests in almost any mainstream programming language, including Python. Let's look at how you can use Selenium, with Python, to automate UI interactions.

To get set up, you'll need to download a web driver for a browser you have installed locally.

You can find webdrivers on listed on the selenium docs[18]. For example, if you're using Google Chrome you can download an appropriate driver directly from Google[19] for your specific Chrome version. Once you've got the driver you can start programming! The test script below will programmatically load a website in Chrome and test the title is as expected:

```
import pytest
from selenium import webdriver

# Module scope re-uses the fixture
@pytest.fixture(scope='module')
def driver():
    # path to your webdriver download
    with webdriver.Chrome('./chromedriver') as driver:
        yield driver

def test_python_home(driver):
    driver.get("https://www.python.org")
    assert driver.title == 'Welcome to Python.org'
```

For a second example, we can click the 'Downloads' link and check it redirects us the correct location:

```
def test_downloads_page(driver):
    driver.get("https://www.python.org")
    link = driver.find_element_by_link_text('Downloads')
    link.click()
    assert driver.current_url == 'https://www.python.org/downloads/'
```

Selenium can perform any action a user might want to take in a browser. You can navigate through web pages, download files, simulate keyboard strokes, and much more. Tools like Selenium are essential for testing applications with web interfaces. Selenium can test arbitrary user journeys through your website (i.e. a true E2E test), but can also be used to check a frontend in isolation, for example by mocking out all backend components. Both approaches are useful, but make sure you know which you're doing, and why!

Headless Browsers

You can do this kind of testing using a headless browser for performance gains. A headless browser is a web browser that doesn't render HTML pages into a GUI. This may seem a little frustrating from a user point-of-view (since it's harder to see that your tests are performing the actions you're expecting them to), but it can speed up UI tests greatly. A headless browser isn't required for automated UI testing – a fully rendered browser can work just fine, but if you're running more than one or two browser tests the impact will be noticeable.

Until a few years ago you would need to find specific browsers designed for headless operations. Fortunately, that's no longer the case and some mainstream modern browsers now have built-in support for headless operations (for example Chrome and Firefox).

To run selenium with Google Chrome in headless mode, you can modify the fixture shown earlier to add the appropriate configuration option:

```
@pytest.fixture(scope='module')
def driver():
    # path to your webdriver download
    opts = webdriver.ChromeOptions()
    opts.add_argument('--headless')
    with webdriver.Chrome('./chromedriver', options=opts) as driver:
        yield driver
```

Running the code this way, you won't see a brief browser window pop up, and the results will come back a little faster. Keep headless browsers in mind if you're considering any significant amount of UI testing.

System tests are a good opportunity to test some non-functional requirements (NFRs) of the system. These tests typically require deployment onto production-like infrastructure so cannot be performed lower down the test pyramid. When we talk about NFRs we mean metrics that aren't part of the core functionality of the software, but have a significant impact on the fitness-for-purpose of the solution. Such things include:

- Resource (CPU / RAM) usage
- Benchmarking (e.g. performance & latency)
- Security (e.g. penetration testing)
- Load testing
- Regulatory compliance
- ...and many more

Chapter 6

Tests and Refactoring

In the previous module we introduced refactoring as the practice of transforming your source code while preserving behaviour. You might do this for a variety of reasons, but the usual candidates are to improve readability, performance and/or reduce technical debt. Occasionally you'll take on larger refactoring projects, where you might be re-architecting the whole codebase, or switching a major dependency. In this section we'll discuss how automated testing enables and encourages safe refactoring.

How do I know I'm not breaking anything?

Concern about breakages is the number one reason why refactoring is often skipped and technical debt builds over time. This makes future refactoring efforts even harder, as the debt builds up and the scale of the challenge grows. The key to keeping refactoring safe is two-fold:

1. Have automated tests to validate system behaviour
2. Refactor little and often

This should sound familiar to the Red-Green-Refactor cycle, where refactoring is enshrined as a key step in the iterative TDD workflow. However, this approach to refactoring is not limited to TDD workflows, and applies to more than just unit tests.

Preserving Behaviour

Refactoring is all about changing code while preserving behaviour. Your tests, if well written, should be testing behaviour rather than implementation details. Does this mean that refactoring should never cause any tests to fail? Unfortunately, the answer is *no*. It all depends on the level at which you're refactoring.

When refactoring, you need to think about what kind of behaviour you are preserving. You should decide this before making any changes, as the level you choose to preserve behaviour at determines the kind of tests you need in place to refactor safely:

- **Function/class behaviour**. Maybe you're adopting a new code style convention, or making use of a new low-level language feature. In this case you are adjusting the internal operations of your functions and classes. This should not break any tests. *Unit tests* in particular will be valuable in ensuring your refactoring is safe.
- **Component (e.g. API) behaviour**. Perhaps you want to re-write a component of your application (e.g. a microservice). You anticipate changing a lot of the classes within the microservice, but want to retain the same external API through which it

communicates to the rest of the system. In this case you will need to write new unit tests and remove some obsolete ones. *Integration tests* will ensure that the component's overall behaviour remains unchanged.

- **Feature behaviour**. In this case you're making (potentially large) changes the architecture of the system. This will break integration tests and you'll need to write new ones, along with new unit tests for any lower-level components that are reworked. *End-to-end tests* will remain unchanged, and can be used to validate that the new architecture is feature-comparable to the old one.

Before embarking on any refactoring task, *ensure there are tests of a sufficiently high level to validate your refactoring*. It's no good to plan a component re-write when you've only got a unit test suite, because the unit tests operate at a lower level and have nothing to say about behaviour of the whole component. If the relevant tests don't exist you should write them before starting on refactoring.

Additional Material

Testing Terminology

This module has introduced loads of names! Unfortunately, software testing is a field with a lot of terminology, and very little of it with standard definitions. We've talked about unit tests, integration tests and end-to-end tests. We've also used the terms "system test", "UI test" and "acceptance tests" to refer to specific types of test, in particular sub-categories of end-to-end tests.

There are many more terms you'll find across the industry! Below are just a few that you may encounter. Be aware that many of these terms are not standardised, and different businesses will mean different things when talking about them.

- **Functional test** - an end-to-end test focusing on a particular slice of functionality. For example, a specific user journey.
- **Service test** - an integration test.
- **Smoke test** - a very quick, simple test to check the system isn't completely broken. Usually run before more thorough tests.
- **Penetration test** - an external test for security vulnerabilities.
- **Load test** - an end-to-end test of the system's response to increased traffic, used to identify bottlenecks.
- **Snapshot test** - discussed in this section.
- **Contract test** - discussed in this section.
- **Behaviour test** - discussed in this section.

Regression Testing

Regression testing is the practice of re-running existing tests to check that no old functionality is broken by recent changes to your software. A **regression** is change that breaks old tests.

With good automated testing it's normal to always run regression tests, so you might not see the term used (it's just 'testing'). In contrast, *manual* regression testing is an extremely time-consuming process, particularly if code changes are frequent.

Snapshot Testing

Some tests are easy to write. Perhaps a function computes and returns a single value based on inputs. You can provide inputs and check for a known result. This is a textbook unit test.

Unfortunately, sometimes tests don't have an easily verifiable result. Perhaps you need to test code that returns a fragment of HTML or a complex JSON object. It can be difficult to create a suitable test result to validate. Even if you did, maintaining it would be very difficult!

Snapshot testing is here to help. A snapshot tool compares your test output with previous outputs, and warns you if it has detected any change. You don't need to provide an expected result - it generates one for you the first time you run the test. Snapshot testing is most commonly used as part of UI testing, but can be applied anywhere.

Below are just a few tools in this space:

- snapshottest[20] - a python package for snapshot testing
- Jest Snapshot testing[21] - a common option in the Javascript ecosystem, primarily for testing web interfaces
- BackstopJS[22] - snapshot testing web interfaces using screenshots!

Contract Testing

Integration tests usually require running multiple services (e.g. a frontend and a backend, or a cluster of micro-services), and checking they can talk to each other. Sometimes it's preferable to test sub-system interactions without actually running them. Contract testing is a way of doing this.

In short, you tell a testing tool what you expect the interactions between two services to look like (the **contract**), and it verifies that both services adhere to the contract. Contract testing not a new idea, but has experienced a recent resurgence as modern micro-service architectures make traditional integration testing more complex.

There are many tools out there to help write and execute contract tests. Take a look at a popular option - Pact[23] - to learn more.

Behaviour-Driven Testing

```
Title: Returns and exchanges go to inventory.

As a store owner,
I want to add items back to inventory when they are returned or
exchanged,
so that I can track inventory.

Scenario 1: Items returned for refund should be added to inventory.
Given that a customer previously bought a black sweater from me
and I have three black sweaters in inventory,
when they return the black sweater for a refund,
then I should have four black sweaters in inventory.

(source: Wikipedia)
```

Does that look like a test? Well it is! Behaviour-Driven Development (BDD) is a form of TDD where acceptance test cases are phrased in a readable, declarative syntax. As the name implies, the test cases focus on application behaviours, rather than any technical details.

The power of BDD is that it enables those with no programming knowledge to read and write test cases, and those test cases can be expressed clearly in business terms. From a developer point of view, BDD is not often popular - BDD frameworks require additional configuration to map natural language to code, and obfuscate the generated code. Nevertheless, BDD is a powerful tool in business environments where non-programmers need to understand your test suites.

Cucumber[24] is a popular BDD framework, and the authors helpful maintain a list of BDD frameworks across many languages. Check it out here: https://cucumber.io/blog/bdd/the-ultimate-guide-to-bdd-test-automation-framewor/.

Laboratory/Scientist

Refactoring code can be especially risky when modifying critical paths (e.g. performance bottlenecks) in a program's code. Wouldn't it be useful if you could test a change in production without risking breaking existing behaviour?

Laboratory[25] is a library for Python that is inspired by GitHub's Scientist[26] library for Ruby. Put simply, it allows you to run an alternative codepath alongside the original, unrefactored code, while testing for changes in return values or regressions in performance.

This technique can be used in combination with **Branching by Abstraction** to safely test moving over methods/endpoints from the abstraction to the new service before committing to the move.

This GitHub blog post[27] goes into more detail on how these tools should be used.

Running Tests Before Review

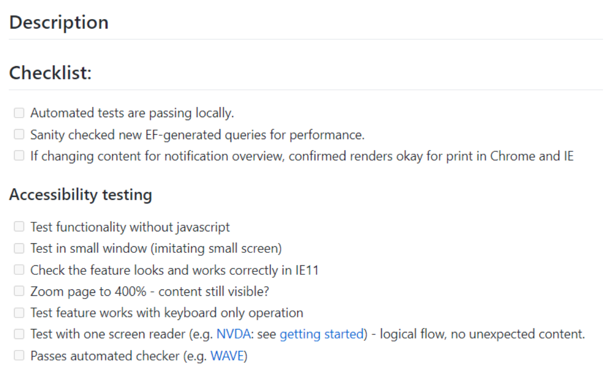
Some projects can have a large number of different test suites that need to be run separately (e.g. Unit, Integration and UI tests).

It is important that any new developers are made aware of all the test suites applicable to the project they are working on.

This could just involve providing instructions in a README file but there might also be a number of manual tests that have to be run (dependent on the task that the developer is working on).

Code Review Templates

Code review templates can be useful for helping developers identify and list the checks that they need to have carried out before submitting a review:



Description

Checklist:

- ☐ Automated tests are passing locally.
- ☐ Sanity checked new EF-generated queries for performance.
- ☐ If changing content for notification overview, confirmed renders okay for print in Chrome and IE

Accessibility testing

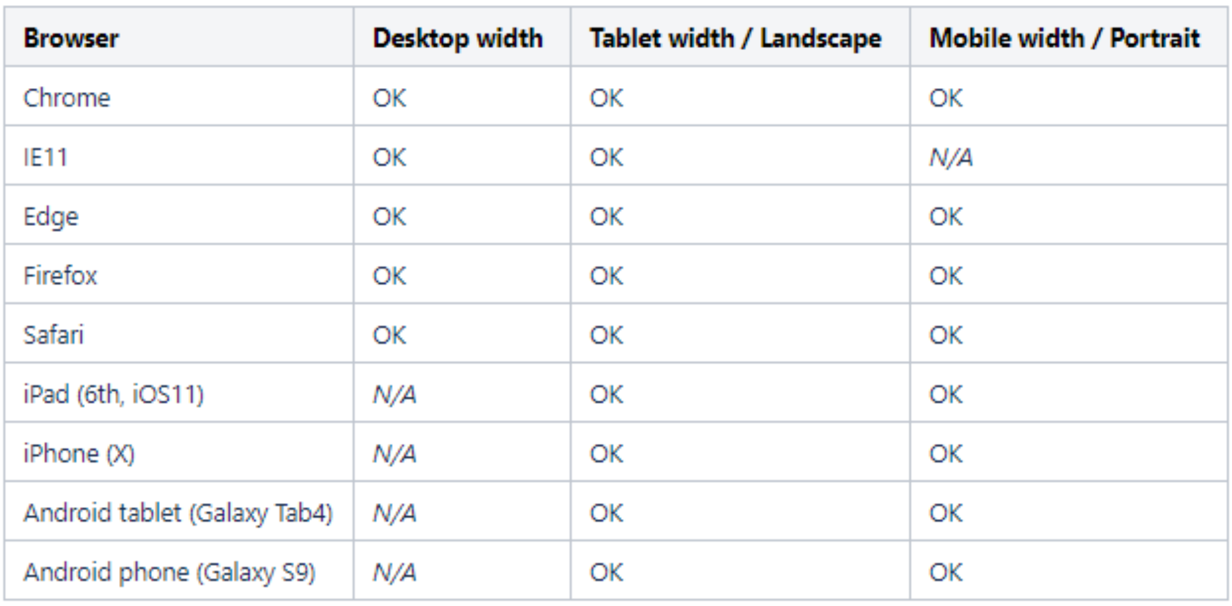
- ☐ Test functionality without javascript
- ☐ Test in small window (imitating small screen)
- ☐ Check the feature looks and works correctly in IE11
- ☐ Zoom page to 400% - content still visible?
- ☐ Test feature works with keyboard only operation
- ☐ Test with one screen reader (e.g. NVDA: see getting started) - logical flow, no unexpected content.
- ☐ Passes automated checker (e.g. WAVE)

Code Review Checklist (taken from public repo https://github.com/publichealthengland/ntbs_Beta)

In GitHub a code review template can be setup by creating a `pull_request_template.md` file in the `.github` folder in the default repository branch (more details about GitHub review templates can be found here: https://help.github.com/en/github/building-a-strong-community/creating-a-pull-request-template-for-your-repository).

This is usually a good opportunity to mention tools that need to be run manually:

- Accessibility: WAVE[28], Screen Readers
- Cross Browser/Device Testing: BrowserStack[29], Browser DevTools
    - Usually you'd normally provide a testing matrix in the code review template specifying which devices/browsers/orientations you'd be testing against:



Browser	**Desktop width**	**Tablet width / Landscape**	**Mobile width / Portrait**
Chrome	OK	OK	OK
IE11	OK	OK	*N/A*
Edge	OK	OK	OK
Firefox	OK	OK	OK
Safari	OK	OK	OK
iPad (6th, iOS11)	*N/A*	OK	OK
iPhone (X)	*N/A*	OK	OK
Android tablet (Galaxy Tab4)	*N/A*	OK	OK
Android phone (Galaxy S9)	*N/A*	OK	OK

Links & References

1. https://en.wikipedia.org/wiki/Caesar_cipher
2. https://docs.python.org/3/library/unittest.html
3. https://docs.pytest.org/en/stable/
4. https://docs.nose2.io/en/latest/
5. https://docs.pytest.org/en/stable/
6. https://docs.pytest.org/en/stable/goodpractices.html#test-package-name
7. https://docs.pytest.org/en/stable/reference.html#confval-python_functions
8. https://docs.pytest.org/en/latest/fixture.html
9. https://pypi.org/project/pytest-cov/
10. https://docs.python.org/3/library/unittest.mock.html
11. https://docs.pytest.org/en/latest/monkeypatch.html
12. https://docs.pytest.org/en/latest/monkeypatch.html
13. https://docs.pytest.org/en/stable/
14. https://www.sqlite.org/index.html
15. http://www.h2database.com/html/features.html
16. https://github.com/testcontainers/testcontainers-python
17. https://www.selenium.dev/
18. https://www.selenium.dev/documentation/en/webdriver/driver_requirements/
19. https://chromedriver.storage.googleapis.com/index.html
20. https://pypi.org/project/snapshottest/
21. https://jestjs.io/docs/en/snapshot-testing
22. https://github.com/garris/BackstopJS
23. https://docs.pact.io/faq/convinceme
24. https://cucumber.io/
25. https://github.com/joealcorn/laboratory
26. https://github.com/github/scientist
27. https://github.blog/2016-02-03-scientist/
28. https://wave.webaim.org/
29. https://www.browserstack.com/