Course Overview

### Testing in Python 3

by Emily Bache

<https://app.pluralsight.com/library/courses/python-3-testing/table-of-contents>

Course Overview

Hello. My name is Emily Bache, and welcome to my course, Testing in Python 3. I am a technical coach at my own company, Bache Consulting. I'm really excited to be able to show you some of my favorite testing techniques in this course, like test‑driven development, approval testing, as well as some strategies for getting difficult code that has awkward dependencies under test. This course is going to equip you to write automated tests for your Python code. Some of the major topics that we will cover include unit tests and how you should design them, how to use Pytest, a popular and flexible testing tool, when and how to use mocks and other kinds of test doubles, and useful techniques like parameterized testing and monkey patching. By the end of this course, you'll know how to design tests for your Python programs that are useful and maintainable for the long term. Before beginning the course, you should be familiar with writing Python programs using standard library modules, as well as designing your own classes and functions. I hope you'll join me on this journey to learn about test design with the Testing in Python 3 course at Pluralsight.

Unit Test Vocabulary and Design

Introduction

Hello and welcome. My name is Emily Bache, and I'm a technical coach. In this module, we will go through unit testing vocabulary and design in Python. Before we get started, you should know which versions we're using in this course. This course was created using the displayed versions of Python and pytest. This slide shows the versions of Python and pytest for which the information in this course applies. In this module, we'll be looking at the basic syntax and execution of test cases and explain the vocabulary we use to describe automated unit testing. It helps you to have words to describe what you're doing. I'll be illustrating the theory with a simple example using the unittest module, which is included in the standard Python distribution. Then I'd like to look more closely at how to design test cases, how to structure the test case code. This is perhaps the most important thing for successful automated testing. We're discussing unit testing in this module, that is tests for small pieces of code that are part of a larger system. It could be a method or a function, a module or a class, or a small group of related classes. An automated test is designed by a person, usually the same developer who develops the code that's being tested. But when you run it, you shouldn't need to do anything. It just runs and then reports either pass or fail. Mostly, this course is about unit testing. By a strict definition of unit test, it's not a unit test if it uses the file system or a database or the network or, in fact, any other slow external resource. By testing tiny units that run in memory, you get simpler tests that run very fast and still prove something useful. Those other tests might also be useful, but I think it's important to know the distinction.

Demo: Writing a Test Case Using unittest

I'd like to demonstrate how to write some simple unit tests for a Phonebook class. This is a description of the problem we'll be solving. Given a list of names and phone numbers, make a phonebook, that is a kind of catalog where you can look up somebody's phone number if you know their name. The tricky part is to work out whether a given phonebook is consistent. Now, a phonebook is consistent if no number is a prefix of another number. You could have a phonebook with these two numbers for Bob and Anna, and that would be fine. But if your list also included this third number, you'd have a problem. You wouldn't be able to phone Bob. So you could say Bob and Emergency are inconsistent, and this phonebook wouldn't be any good. Here I've got an empty project for my phonebook code, and I'm going to begin by creating a file to keep my test cases in, and I'm going to name it test\_phonebook. The test\_ is the prefix that the testing framework will look for when it's looking for unit tests. So it's a good naming convention to follow. And similarly, I need to declare a class that I also use a naming convention. It's going to be called PhonebookTest, and I need to inherit from unittest.TestCase. So there's the the test class where I'm going to put my test cases. And each test case also needs to follow this naming convention. It needs to start test\_. And then the next part of the method name here should be the scenario that I want to test. But actually for the moment, I'm just going to start by putting a placeholder there because I want to just check that my testing framework is set up correctly. I just want to check that I can actually execute it. And I do that by executing the Python with the unittest module, and it says it's run 1 test in like 0 seconds, which is very fast because my test didn't do anything. And it said that it's passed. Everything is okay. So that's reassuring. My test framework is set up correctly. Okay, so let's try and see if we can replace this with an actually more meaningful test for my phonebook. Now the test case that I had in mind to start with was looking up by name. That's the kind of core functionality of a phonebook. You can look up people's phone number if you know their name. So let's start by creating an instance of my Phonebook class. Now, it's been marked in my editor as underlined in red because, of course, that class doesn't exist yet. This is a completely new project, and that class isn't there. So I'm just going to use my tools to create that class. And by default, it puts it in the same file. But of course, this is going to be my production code. So I want it in a different file than my unit tests. So I'm just going to move it into its own file. And I thought a file called just phonebook would be the right kind of place for it. So there we are. The tool moved it to a different file, and now I'll just open that up so that we can see the test and the code side by side. Okay, so the next thing is I want to actually start using my new Phonebook class. I want to be able to add an entry. Let's put in Bob with a very simple phone number. And I just need to go and create that method in the production code and make sure the parameter names are suitable. Okay, so now we've added Bob, we should be able to look him up and find out his number. So let's try that. And of course, the lookup method doesn't exist either. So I'll just go and create that. And at the moment, I'm not writing any implementation for these methods. I'm just creating the interface to my Phonebook class because I want to write my test case so I can see how the class is used before I invest too much effort in actually implementing it. So, there's one part of the test left that I need to write. That's the assertion. And there are lots of ways of doing assertions in this test framework you can see here, but I'm going to choose assertEqual because I want to check that number is the one that I expected, the number that I put in earlier. Okay, so now my test case that has all the parts necessary that I could execute it. And I'm totally expecting this test to fail because, of course, I haven't actually implemented my Phonebook class yet. Great. So my test has failed and the framework has told me very clearly that this has failed. The name of the test is clear there. We've got a stack trace pointing at the assertion that's failed. And the error message is we were expecting 12345, and we've got back none, which is what you'd expect when your implementation is entirely empty.

Test Suites and Test Runners

Let's go through the unit testing vocabulary of what we've just done. A test case is fundamental to automated testing. Each test case should exercise a unit of code and check it works correctly. You should be able to run it independently of any other test cases you may have as many times as you like, and it should always clearly report whether it passed or failed. You may have several tests for the same unit, but each should be independent and not have side effects that other tests either rely on or will be caught out by. For example, you shouldn't have one test create some data that another test then uses. A few times now, you've seen me use a test runner. This is a program that executes test cases and reports the results. With the unittest module, the command line test runner is built‑in. If I run python ‑m unittest, it will automatically discover and run all of your test cases. Many IDEs also have a way to run your tests. They have their own built‑in test runner. Here I'm creating a new run configuration in PyCharm. I just need to point out where on my disk it should look for my test cases, and then it will run python ‑m there in its own built‑in test runner. It looks fairly similar to the command line interface, except it has this tree view on the left where you can view all your test cases. When I only have one, it doesn't make any difference really. But if I had several, it's more convenient for browsing all the different test results. When you're working interactively in an IDE like PyCharm developing code and tests, it's probably convenient to use the built‑in test runner. If you're instead running your tests unattended in a continuous integration environment, for example, then a command line tool is more useful. The point is though you have this separation of concerns, the test case is defined independently of how it's run. You should be able to design your test case without worrying about which test runner will be used to execute it. Okay, I think it's about time we worked on getting this test to pass. So I'm changing the production code here, and I'm going to start using a dictionary to store the phone numbers in. I'll need to initialize that in the constructor. And then in the lookup method, I can get the name that they're looking for. Okay, so my test runner here in PyCharm is showing the test results are good. I get a green tick. And it says ran 1 test. Everything is OK. By default, it actually hides the passing tests, but you can show them by clicking this button. In the same way, we could use the command line test runner to run these tests. And again, it hides a lot of the information. It just tells you the important thing. Everything is okay. Everything's passing. I think it's time to add a second test case. I want to see what happens when I look up a name that's not present in my phonebook. I think it should raise a key error. I'll declare a new test method in the same class, PhonebookTest, and I'll call it missing\_name. I'll still need to construct a phonebook. And then this time, I'm going to use assertRaises because I want it to raise a key error. It's a little bit different than the assertEquals method. I need to put it in a context manager. This means that unit tests will ensure that the lines enclosed in the with statement will throw a key error. And if they do, the test will pass. Unfortunately, my test doesn't pass. It turns out that it doesn't raise a key error, and that's because I'm using the get method on the dictionary. If I instead use an ordinary dictionary lookup, that will have the behavior I want. That will throw a key error when it's missing. Now that we have two test cases, we can actually start to talk about having a test suite. A test suite is simply a number of test cases that are executed together by a test runner. In our example, we have two test cases defined in the same class. But more generally, a test suite will be made up of test cases in several different classes for different units. It's about separation of concerns. You can design your test case without worrying too much about which test suite it will end up being a part of and design your test suite without worrying about the test runner that will execute it.

Demo: Test Fixtures - Using setUp and tearDown

Let's continue with the demo. We've got the basic lookup functionality working now, so it's time to start working on the next part, checking for consistency. If you remember, a phonebook is inconsistent if any of the numbers in it are the same or if one is a prefix of another. The simplest example I can think of is for an empty phonebook, which, of course, should be consistent. Let's write a test case for that. So I'm adding a new test case that's going to check an empty phonebook is consistent. And I need to begin by constructing a new phonebook instance, and then I can check it is consistent by calling this method on it, is\_consistent. Then I can assert that that is going to be true. So that's a different assertion we haven't seen before, and this one is for checking a Boolean, which is exactly what we need here. So, I'll go and add that method in the production code and run the test, and it fails with a very clear message. None is not true. So at this point, the very natural move would be to go and start to implement the is\_consistent method. But I've just spotted something that I've got distracted by, and I can see that this test is similar to the others. This first line that constructs an empty phonebook is shared. All the other tests do that too. So I want to take this opportunity to show you a few more features of this unit testing framework. While I'm adding some setup code to construct that phonebook, I'm just going to mark this currently failing test as a work in process so that I don't get distracted by it while I'm working on refactoring the tests. So I add this annotation to it to say that I want to skip this test, and that the message is, well, it's Work In Process. I'll come back to it soon. So now when I run the tests, it clearly marks this test as skipped. So, with everything looking good according to unittest, let's start work on that setup method. Now, it's one of the methods that's inherited from the unittest.TestCase class. So I'm just going to ask my IDE to help me to override it correctly. The setup method is a special method in TestCase because it is called before every test case is executed, which means that you can construct things here that are fresh and available for each test case, but without duplicating all that code. So here I'm going to use it to construct a phonebook and put it in a data member of the class that then I can use that instead of instantiating it in every test case. So here, I can use the data member in this test case and this one as well and in the third test case. So it kind of looks like they're all sharing the same instance of phonebook. But actually, that's not the case because a new class is instantiated, and it calls setup again before every test case. So this is really just a way to reduce code duplication. If I run the test again, they all look good. Still the same as before. So that seems to have been a successful refactoring. I just wanted to show you there's a second special method in TestCase, which is called tearDown, and this one is called after every test case method. So it allows you to release any resources that you tied up in the setup method. In this case though, the phonebook is entirely in memory. And at the moment, there's no resources we need to release. In the future, that might change though. So I think it's good to have the tearDown method there to remind us. Okay, so now I can get back to what I was working on before, that little refactoring. I can take away the skip mark and look and see I've got my failing test to remind me what I was working on. And here I need to actually do some implementation in the is\_consistent method. So, that seems to work now. It was all I needed for this test case at least, but I'm sure there's more work to be done. I'm just looking over the code again. I'm just spotting some opportunities to improve the formatting. But otherwise, I'm feeling quite good about this code so far.

Test Fixtures - Execution Order

This is a good moment to explain some more vocabulary. In this test class, we used a setUp method to reduce code duplication in our tests. It's an example of a test fixture. A test fixture is supporting code that creates and configures resources needed by test cases and will clean them up afterwards. In the unit test framework, the test fixture actually comprises two special methods, which you override in your test class, setUp and tearDown. These are shared by all of the test cases found in the same test class. And here we have three test cases, all with the same setUp and tearDown. This is the order the methods execute in. First, setUp, then your TestCaseMethod, and then tearDown. In a strict unit test where all the objects are in memory, there is often no need for a tearDown method since these objects will be released by the memory manager. TearDown is useful though if you need to release other kinds of resources, for example a database connection or to delete some temporary files. It's important to understand what happens on test failure. Even if an exception is thrown in your test case or an assertion fails, tearDown will still be run. The test will, of course, fail because of the exception, but your tearDown will still be run before that happens. And the reason is because tearDown is supposed to release the resources allocated in setUp, and this still needs to happen even if the test has failed. If, however, there is an exception thrown in the setUp method, the test will fail. In fact, the test method will not be called. But more importantly, tearDown won't be called either. That can catch you out. The reason though is that if setUp has failed, then the test framework assumes that you didn't get to allocate the resources you intended to, and there will be nothing for tearDown to release. I've introduced several major pieces of vocabulary in unit testing, the unit under test, which is the piece of code that we want to check, the test case, which will go through a particular scenario or use case for the unit under test, a test suite, which is a collection of test cases that are executed together using a test runner, which runs the test cases and keeps track of which ones are failing or skipped, and finally, the test fixture, which is supporting code for managing resources needed by your test cases. In the next sections, I'd like to zoom in on the test case and talk about good test design principles.

Demo: Test Case Design - Names

We're going to do some more work now on the phone numbers exercise and talk more about test design. We need to do some more work on the is\_consistent method, so I want to write a test case that will force me to do that. And since we're talking about test design, I wanted to show you a less good way of doing that. I'm going to declare a new test case called is\_consistent, and I'm going to put an entry in it. I can use the same kind of entry that I had before for Bob. And then the phonebook should be consistent at this point with one entry. It should also be consistent if I add a second entry with a consistent phone number. So now I know that my phonebook could handle two phone numbers that are consistent. And now it's probably time to add a third phone number that is not consistent with the others. Let's add Sue with the same phone number as Bob. So Sue's number is identical to Bob's, and the phonebook should not be consistent, and we'll use assertFalse to check that. That's not the only kind of inconsistency there is though. There's also the case where a number is a prefix of another, and that should also cause the phonebook to be inconsistent. Okay, so this test case now covers the major use cases for the is\_consistent method, all the different ways that it could be consistent or not consistent that I can think of at the moment. But I said at the start that this was an example of poor test design. And hopefully you'll be able to see why when I run this test. The new test has failed, which is expected at the moment. But just look at the information we've got. If this test failed again in the future unexpectedly, this is what you'd have to go on. Now the most important job of a test case, of course, is to fail when there is a problem. But the second most important job is to make it as easy as possible to understand what the problem is so you can fix it. The first thing you see is the name of the test that failed, is\_consistent. That actually isn't too bad. It tells us the problem is in the is\_consistent method. Beyond that though, it's pretty vague. In general, we want the names of our test cases to summarize the scenario being tested. So we've got look up by name, missing name, consistent when empty, and each of these already is describing a particular use case or scenario for the class under test. So rather than just having another test case called is\_consistent, it would be clearer to have cases for all of the major scenarios, consistent when all different, inconsistent when duplicates, inconsistent when duplicates prefix, something like that. There's another problem with having three scenarios in one test case. The assertion that failed is here in the middle of the test case. As soon as that happened, an exception was thrown, and we can see the error message here. But that means that the rest of the test case didn't execute. We don't know whether the following lines of code would also have contained further errors. When you have an unexpected test failure, you want to be able to quickly work out what the problem is. And if the test is written like this, we don't get all the information about all the scenarios. In a unit test, this is the standard design pattern. A test has three parts, a range where you set up the object to be tested and its collaborators, act where you exercise the unit under test, and assert where you make claims about what happened. The first test case we wrote uses this pattern. The first line arranges a phonebook with one entry. The next line acts to test the lookup method. And the final line asserts the number we got was correct. The test we just wrote has a more complicated structure. The first line arranges a phonebook with one entry. And the next line both acts to check consistency and asserts on the results. That's fine actually putting act and assert on the same line of code. The problem here is that the test just carries on with more arranging, another entry in the dictionary, more act, more assert, and it just goes on and on. This behavior that it stops executing the test on the first assertion failure is actually a deliberate design feature of this testing framework. Other test frameworks may allow you to keep going and execute a whole test case even if one of the checks fails. But for unit tests, they should be small and fast, and the one assertion should be at the end. This is the tools way of encouraging you to make several small tests rather than one big one with several scenarios.

Demo: Test Case Design - Arrange - Act - Assert

Let's look at a better way to design tests for the is\_consistent functionality. We're going to replace this one big test with several smaller tests for different scenarios. So is\_consistent\_with\_different\_entries is the first scenario. And for that, we need two entries, and then it should be consistent. Then we want to test the scenario where it's inconsistent because we have duplicate entries. And for that one, we're going to need Bob and Sue because they have the same number. And I've got a little comment here that says that they're identical, and I don't need that anymore. The test name is communicating that much better than a small comment hidden in the middle of the test case. And then there's the third scenario that I want to test where it's inconsistent because there's a duplicate prefix on one of the numbers. And for this one, Sue has just part of Bob's number. Okay, so that replaces that big test with three smaller tests, and each of them follows the arrange, act, assert structure. And when we run them, we get more information than we had before because now we have two failures instead of just one big test failing, and it's telling us clearly that the detecting inconsistency is not working. Okay, so I'd better go and implement that then. So I'm going to go for a fairly brute force kind of algorithm. I'm going to iterate over the items in the dictionary twice. And then of course, at some point, I'll get the item that's the same as itself in which case I should just continue. But if I find the numbers are too similar, then I should mark this as inconsistent. Okay, let's see if that works. Yes, it seems to. This algorithm is quite simple, but it seems to do the job. If we review all the code and tests we have so far, we can see we have some basic functionality working, but we're probably not done yet. As well as these unit tests, you usually need some larger‑scale tests, probably performance tests. We're not going to do that now. Before we leave the unittest module, I want to draw your attention to the documentation where you can find out more. This part lists all the assert methods that are available. In our example, we used assertEqual, assertTrue, assertFalse, and assertRaises. As you can see, there are several other options.

Module Summary

In this module, we've gone through some basic testing concepts and vocabulary illustrated with the unittest module. We've designed some test cases, run them with two different test runners, and placed tests together in a test suite. We've designed a test fixture for shared setup code. At their core, all of these design elements are about separation of concerns, making it easier to create flexible and reliable test cases. We've also talked about good test case design. In particular, the names of each test case should reflect the scenarios being tested. Within a unit test, you normally have three parts, arrange, act, assert. Writing many small tests, each following this advice, will give you good information about the problem when the tests fail.

Using Pytest

Introduction to pytest

In this module, we're going to learn about using pytest. According to some surveys of Python programmers, pytest is used more widely than the unittest module we looked at earlier. We will learn how to define test cases with pytest, how to interpret the output you get when a test fails, how to avoid duplication in your test code by using test fixtures and parameterized tests, and how you can organize your tests in a larger project, including control of which ones are run. Originally, the Python unittest module was a port of JUnit for Java, which was created by Kent Beck and Erich Gamma back in 1997. Unittest was included in Python version 2.1 in 2001. And at the time, JUnit was an emerging standard and had been translated to many programming languages. All of these ports became known as xUnit frameworks because they all worked in basically the same way, and it was easy to move between them. The thing is since then, both JUnit and unittest have moved on a lot, and they're no longer all that similar. That led some people including Holger Kregel to create an alternative that was more native to Python. Pytest is a powerful and popular testing framework. It's not a member of the xUnit family, and it's not in the standard Python distribution. There are instructions for how to install pytest in the pytest documentation. Perhaps the easiest way to do it is with the pip installer. If you haven't used pip before, you should probably look up the Pluralsight course, Python Fundamentals.

Demo: Writing Unit Tests Using pytest

I'd like to demonstrate how to use pytest using the same example that we saw in the first module. We'll write some unit tests for a Phonebook class. This is a description of the problem you'll be solving. Given a list of names and phone numbers, make a phonebook and determine if it is consistent, i.e. if any number is a prefix of another. Here, I've created an empty project with some folders ready for putting source and test code in. And I've even prepared an init.py for my tests module. So I'm going to create a new Python file for keeping my tests in called test\_phonebook. And here, I can write my first test case. I'll start just with a placeholder so I can check that my framework is working properly. So here to try and run my test, I need to call Python with the pytest module, and that seems to work. So now I feel confident that I could actually add a test that actually does something. So let's take the same test case as before, lookup\_by\_name. I'll need to instantiate a phonebook instance, and I'll just create that one here for the moment. Let's just check the tests still pass. Yep. Maybe I should just move that to its own module under the src folder, which is where I want my production code to live actually. Now after I made that move though, my test code can no longer find the production code. And the easy way to fix this is just to set the Python path on the command line. It might be even more convenient to use the test runner that's in PyCharm. So I'm just going to add a test runner here with pytest, and it's just going to add the content roots and the source roots to my Python path automatically for me so it's a little bit more convenient. So I can just run it there. And yes, that seems to be working. Okay, so let's write the rest of the test case. I'll want to add Bob with a known number and add that method to my Phonebook class. So I'm just going to rearrange my windows so I can see that at the same time. Give it some decent parameter names. And then when I've got a number there, I should be able to look it up by name, and that also needs a good parameter name there. And then I can assert that the number is the one I expected it was. So there, that's what a test case looks like in pytest. The assertion here is just using the standard Python keyword assert. When I run that, it fails, and I can see a very clear error message here. The test result here has shown me which test has failed, what the expected and actual values were. It's also printed out all the source code in the test case, so I can see the context of the failing assertion. And it's very clearly said the thing on the left of the equals equals was none when I was expecting it to match the thing on the right, which was this string 12345. So that's a very clear failure message there from pytest. So I'm just going to add the small amount of production code that's needed here to make this test pass. I'll need to store the numbers in a dictionary, and I'll be able to use that to both store it and look it up. So now the test is passing. Let's look back and compare this new test case with the one we wrote in the first module using unittest. Of course, the actual code is very similar, but it's the code needed to satisfy the test framework that differs. For the unittest version, we need to have a class and inherit from TestCase. As well as following the test\_ naming convention for test cases, you're expected to use methods like assertEqual, which don't follow Python's PEP 8 style guide. Actually, they predate PEP 8. In the pytest version of this test, we didn't have to import or inherit anything. We just had to follow the test\_ naming convention. We also didn't need a specific assertion method. We were able to use the built‑in Python assert functionality. Let's review again the failure message that you get for this test, here shown in the command line test runner. Pytest is working really hard under the covers to work out what was going on when that assertion failed. It's doing some really clever bytecode manipulation. But to use pytest, you don't really need to know any of that. Pytest gives you this really simple interface, and then it does its utmost to report test failures so well that you, as a programmer, can immediately see what's happened and why. As well as the command line runner I showed you, you can use the test runner in PyCharm. So of course, when you write your tests with pytest, you don't have to worry about which runner will be used to execute it. It's the same separation of concerns we saw earlier with unittest.

Demo: More Kinds of Assertions in pytest

We'll continue the phonebook demo now and look at using more kinds of assertions in pytest. I'm going to write a new test case for a new method, all\_names. The all\_names method on the phonebook will return a set of all the names that it knows about. So I'm going to put Bob into the phonebook so that this list will not be empty. And so when I call all\_names, I should get back a set containing Bob, but I'm going to put missing as well just so that I get a test failure and I can show you what it looks like when that happens. So I'm going to put the implementation in all\_names as well. It's going to return a set of the keys of the underlying dictionary. So as expected, the test fails, and you can see the error message here is actually very clear. We were expecting Bob and missing, but we actually only got Bob. And again, it's printing out the whole text of the test case to give you the full context of what's going on. I'm going to do it again in a slightly different way just by using the in keyword. I can also assert that this element is in that set. Again, it gives me a good error message, although the PyCharm test runner has confused the expected and actual. But the full output from pytest shows clearly missing is not in Bob. Let me just put this back so that the test is passing. I don't need both of those assertions. I'm going to keep the slightly stronger one that checks the entire set. Okay, let's have another test case with another kind of assertion. I'm going to do the missing name, which is going to try and look up a name that is not present in the phonebook. And if you remember, that should raise a key error. So I'm going to use the pytest.raises functionality here. So at this point, I did actually have to import pytest for the first time in order to get access to the pytest.raises function, which I use in this context manager. And when I run this, it should actually pass. So let's just make a small change so that it fails. I'm going to put Bob in there, so it's no longer missing. And the error message here is very clear. It did not raise a key error.

Demo: Test Fixtures in pytest

If you remember a test fixture is code to manage resources that are needed by the unit under test and to clean them up afterwards. The same fixture code may be shared by several test cases. This is the example using unittest. The setUp and tearDown methods are the fixture. They're methods you override from the super class. Pytest works quite differently. What happens is the test case indicates the resource it needs by specifying in its argument list. Pytest looks for a function decorated as a fixture with the same name and then hooks it all together at runtime. It's a kind of dependency injection. The test just says what resource it requires. Then at runtime, pytest will find one and inject it into the test. Let's look at an example. This is our phonebook example. We have three test cases, and all of them use the same resource. The first line of each test is creating an empty phonebook. We'd like to extract this as a test fixture that could be reused in all the tests. I declare a new function with this annotation pytest.fixture, and the name of the function should be the name of the resource that I'm going to provide. Then, to use the fixture, I need to add the name of the function as an argument to all the test methods that need it. So that seemed to work. All my tests are now getting this fixture. So the name of the function is the convention here it needs to match. If I make a typo in the name of my fixture, the test cases won't be able to find it. You can see here, I've got a clear error message from pytest. I couldn't find this fixture, and that's because of the typo in the method name. It also very helpfully says a list of the test fixtures that are available, more on that in a minute. For the moment, let's just remove that typo. This fixture function is the equivalent of unittest's setUp method. But what about tearDown, resource cleanup? Of course, most of the time in unit testing, the memory management will take care of the resources after the test is executed. Let's change this code a little bit. Imagine that a phonebook was storing some entries in a file cache, and the file should be removed at the end of the test case. And there's a clear method on the phonebook that would do that that we need to call. If I just leave the fixture as it is and run the tests, we can see that now it's creating this file that gets left lying around, and we probably don't want that. So let's modify the fixture so it will be able to clean that up. The first change is to switch the return statement to a yield statement, and that means that you can put some code afterwards that will be executed after the test case. So now when we run the tests, that annoying file is no longer there. That file was created though every time we ran the tests, and that might not be what we want either. It would be better perhaps if the phonebook had a temporary directory to store that file in, and we could pass an argument to the constructor to tell it where its temporary storage should be, and it would put the file there. Now, that looks good. But of course, now our test fixture needs to be able to provide a temporary directory when it constructs the phonebook. And this is where pytest's built‑in fixtures come in handy. There are a whole bunch of test fixtures built in to pytest, including a temporary directory. I can just chain my fixtures like this by adding an argument to the fixture function with the name of the resource that my fixture depends on. So now this is all working, and that temporary file is being put in a temporary directory. If I run this on the command line, I can see all the fixtures that are available in pytest. And my new fixture that I've defined is, of course, on the list, and it's told us we don't have any documentation so far. I'd better fix that in a minute. But here on the rest of the list, I can see lots and lots of useful fixtures available in pytest, including tmpdir, which, of course, will do what I said, return a temporary directory which is unique to each test function. So let's just go and fix that docstring. It's always good to document your work. And now pytest knows how to document my new phonebook fixture. I think this is a very nice way to do fixtures. Both the tests and the fixtures themselves can request resources, and they'll be connected up at runtime. You can document your custom fixtures in the same way as the built‑in ones so it's easy to see what's available for your tests and reuse shared setup across several test cases. It's a loosely coupled, flexible way to handle it.

Demo: Parameterized Tests in pytest

Using test parameters is a way of handling another kind of duplication in test code. Let's look at how that works. If you remember, a phonebook should have an is\_consistent method, and there were three particular scenarios we should write tests for. If all the entries are different, then the phonebook should be consistent. Let's add a skeleton implementation of is\_consistent and check this test fails properly. Yes. And now we can add a basic implementation that will get this test to pass, and we should write tests for those other two scenarios. A phonebook is inconsistent if there are duplicate entries or if one entry is a prefix of another. I'm just going to paste in the same implementation that we saw before. This is a perfectly reasonable way to write tests for these three scenarios. There is some duplication here though. The workflow is the same in all three tests. Add two entries and check if the phonebook is consistent. We could combine all of this code into one test case, but then it would no longer be tested as three separate scenarios. They wouldn't be isolated. A better alternative is a parameterized test. I'm going to declare a new test case just named is\_consistent, and I'm going to add this annotation pytest.mark.parameterize. This annotation takes two arguments. The first is a string describing the additional parameters that your test case needs, and the second argument is a list of data to use as those parameters. I'm going to start with a list with just one element describing the two entries that should be put in the phonebooks. We've got Bob and Anna, both with different phone numbers, and then this should be consistent. So now I've added those additional arguments to this test case that will be supplied by pytest at runtime. I can use them to add entries to the phonebook and assert the result I'm expecting. So that is now equivalent to this first test case, is\_consistent\_with\_different\_entries. I should probably add the data for the other two as well. Okay, well that's passed straight away. And here, if I look at all the passing tests, I can see that this test is consistent. It actually has three under test cases within it. And they are named a little bit strangely, but that's basically the three arguments that are being passed to the test case. This might be a little easier to understand if we had a failing test. I'm just going to modify the code, so it only detects duplicates and not prefixes. So this test now fails. We can see that the error message actually shows clearly what the arguments were when this test case ran. This was when we had Bob and Sue, and Sue has a prefix of Bob's phone number. It's the same test case as this other one that we named test\_inconsistent\_with\_duplicate prefix. The error message here is similar. But this one, to be honest, it is more complicated to understand the error. So there is a kind of tradeoff here between having concise tests and understandable error messages. We've also lost the name of the scenario. We've only seen the arguments, the specific arguments now. In this case, I think it's actually okay. I think we can understand what's going on, and I'm going to remove the three separate tests and just keep the paramatti one. In general though, if you find that you have a lot of parameters or the error message gets difficult to understand, you should probably fall back to having separate test cases with good names. Like test fixtures, parameterized testing is another way to reduce duplication in test code, specifically when the act step or the workflow is the same in several test cases and the difference is what data you pass. You can vary that data via arguments to the test, and those arguments are specified in a decorator, pytest.mark.parameterize

Demo: Organizing Test Code in a Larger Project

So far, we've been working with only small amounts of code. Let's look at how to organize your test code in a larger project, where to keep your test modules and the documentation, and how to control which tests are run. I've made an example project of how the phone numbers code might look if it grew and gained more functionality. I've structured it according to the recommendations in the Pluralsight course, Core Python: Organizing Larger Programs. When you open a new project, the first thing to look at is the README file. This explains what the project is all about and how to use it. Here it tells us clearly, you should probably use a virtual environment and then install the module before using it. It also explains how to run the self tests. All the source code is stored in a module under the src folder, and all the tests are stored under the tests folder. These are the unit tests we recognized from before. The test fixture is in this module conftest. This is a special module that pytest knows about. All the tests in this folder and all the subfolders will look for test fixtures in this file, so it's a good central place to keep test fixtures. We've also got an additional test here with some more realistic test data. This file has got several 100 phone numbers in, so we can do a kind of load test. Another important file for pytest is pytest.ini, which has options that apply to all of the tests in this project. We can run all of the tests on the command line in the usual way. I don't know if you noticed, but that load test with large books was rather slow, and we might not want to wait for that all the time while we're developing. That's where this mark comes in. We've told pytest that this test has the slow marker. So if we pass this flag not slow to pytest, it won't run this one or any other tests with the same mark. Like many things in pytest, this is actually just a naming convention. If I make a typo in the mark on one of my tests, it might not detect it. Fortunately though, as you can see, I've protected myself from that by one of the settings in my pytest.ini. I've told it to always run with strict markers, which means it will only detect markers that are on the list. There are also some markers which are built into pytest. You can get a list of them all by passing the markers flag. The first one on the list there is the one that I've defined, but there are plenty of other useful markers here. One I use more or less frequently is the skip marker. So I'm writing a test for a new feature that this test should fail. I'm going to assert false, but I'm going to add this mark.skip with a comment work in process, new feature. And then when I run all my tests, it's marked with a little s and says that all the tests passed, but one was skipped. If you want to run or not run a particular subset of tests, you can use the pytest.mark annotation. We use this to emit slow tests, but other reasons could be you want to run a particular set of tests before check‑in or as part of your build pipeline. We also showed the mark.skip, which means the test will not be run and will not fail the test suite. It's a good idea to write a comment to remind yourself and other programmers why a test should be skipped. A related mark is skipped if where you give it a conditional to decide whether to run it or not. In this example, it will skip the test if you're running with an older Python version.

Module Summary

To summarize, in this module, we've looked at how to define test cases with pytest. It has a very simple syntax based on a naming convention. We've looked at how to interpret test failures with the analysis you get from pytest where it wants to give you the best possible information about what happened with minimal syntax. We've talked about a couple of ways to avoid duplication in test code by parameterizing tests and sharing resources via test fixtures. Lastly, we talked about organizing test code in a larger project and how to control which tests are run.

Testing By Developers: Why and When

Module Introduction

In this module, we'll step back and look at how testing fits into development work. Some developers are lucky enough to work with other professionals who also write automated tests, testers or quality assurance people. That can be a huge help. In this module, we'll look at why and when developers should write tests for the code that they write. So, why should developers write tests? How can you incorporate writing tests into your personal development process? We'll talk about whether you should do test‑driven development or write the test in some other way. In the last part of this module, we'll talk about how testing can help when you're collaborating with other developers and in particular when you're practicing continuous integration. So let's start by answering that first question. Why do developers write automated tests? Well, of course, there are lots of reasons, but I think there are two most important ones. Firstly, to ensure the code does what you think it does, so you don't embarrass yourself giving people code that doesn't work. Secondly, to protect yourself and others from future regression, that is when things used to work, but then they stop working. Automated tests should detect that kind of problem.

Demo: Developer Feedback Loop

Let's look at an example. I've been developing a program to score the dice game Yatzi, and I'd like to check its working. Let's look at an example. I've been developing a program to score the dice game Yatzi, and I'd like to check its working. This is my new project, and I've written a couple of functions already, dice\_counts, which will make a dictionary of how many of each value are in the dice, and full\_house, which will score a given roll in the full house category. According to this variant of the Yatzi rules, you're going score for a full house if you have two of one number and three of another and you get five dice. So I think this code should work, but I haven't tested it yet. One way to do that is using the Python console. I can import the functions and call them with various arguments and check the return values look okay. So let's try out full\_house with some dice that should score. Yeah, that seems to work. And if I change the dice so that they no longer match the rules, it doesn't score. Let's try another roll that should score. This is the highest‑scoring full house. Oh, now that didn't score, and it should have done. Looks like I've got a mistake somewhere in my code. I can investigate this also using the Python console. This test data doesn't seem to work. Let's take a look at the dice counts on that data. Ah. Now, I think this is where our problem is. It hasn't recorded anything for the sixes in that dice roll. Go and check the implementation. Yes, I have an off‑by‑one error. It's actually stopping at five. So now, through my work on the Python console, I've worked out what the problem is, and I can fix it. The thing is, it's a really short step from here to creating unit tests. If you like using the Python console like this, you should take a look at doctest. It lets you copy and paste from the Python interpreter into a docstring in your function that you can then execute in your unit tests. I have another course, Unit Testing in Python, which has a whole module on doctest so you could check it out. Some people like to test their code using the debugger. Here I'm making a small main program and calling my function so that I can then put a breakpoint and step through it. So again, I'm getting quite clear feedback, and I can see in real time what happens when I step through my code. And it's quite clear that there's something wrong with the dice counts. It's even more clear when I go for the same test data as before with the fives and sixes. So using the debugger like this can also give me fast feedback on how my code works. But again, what I'm doing is not that different from writing a test. This main program is a kind of test. It's just not a very convenient one to maintain or share. Let's go ahead and add that unit test. I'll begin by testing full\_house, and I'll use that test case we had before with 1s and 2s. So that should be 8. Yep. And then the 5s and 6s, which should be 28. But there, my bug. That's the feedback I needed. I would like to improve this test design though and introduce parameterized testing. I'll speed through the details of that. We looked at it in a previous module. Well it took me a moment to set up the parameterization, and maybe the original test was good enough. But I do prefer this failure message, and it's very flexible to add more cases. My point is though, writing this unit test wasn't really more work than any of the other methods of testing my code. Writing unit tests as you develop should help you. Does the code do what you think it does? It should also quickly pay back the time invested. The time you invest in designing test cases should be comparable to the time you would spend anyway testing the thing manually using the Python console or the debugger or through the UI if it has one. If writing those tests is hard, that could be a sign your design needs some work. We'll talk about designs that are hard to test in the upcoming module, Code That's Difficult to Test. In any case, automated tests have an advantage over testing your code in other ways. They turn into regression tests

Demo: Regression Testing

Once the code is working and your tests pass, your test suite takes on a wider role. It should help you to detect regressions. A regression is when something that used to work, no longer does. This can happen when you, a colleague, or somebody else is‑‑‑ This can happen when you or a colleague are changing the design or implementing a new feature. Perhaps you don't anticipate that a change in one part will affect another part. Hopefully, your tests will alert you to this. That's the most important job of a regression test. It should fail when there is a problem and not otherwise. If tests keep failing for no good reason, you stop trusting them, and they become useless. A good suite of regression tests should alert you when you have broken something and stay quiet and green when everything is fine. The next thing you want to know is what went wrong and how you can fix it. Ideally, the test failure message will help you and even point straight at the unit where the problem is. The main reason we invest in readable assertion failure messages and your tool including code snippets in the test failure output is to help you, as a programmer, to quickly understand what's no longer working and which bit of code is responsible. Let's look at an example. Here I've done some more work on my Yatzi score, and I can handle all of the categories and even suggest a good category to score a result in. I've also got unit tests for all the functions, and they all pass. I'm considering making a simplification to the code in this all\_dice\_values method. But it's used in several places, and I'm not sure what impact it's going to have. I'd like to use the range function instead of hardcoding all those values. When I've run my tests, I see that it does have some impact. Two of my tests are failing now. It seems that when there are two pairs available in a dice roll, it will now choose the lower‑scoring one instead of the higher‑scoring one. It seems it's important to go through this list in the other order. So that fixes the problem, at least according to the tests I have so far. I might want to do a little more work to check it didn't impact in any of the other places. Having these unit tests has helped me to understand what my code does and protected me when I was making a change that had a consequence I didn't completely anticipate. These are benefits you still get even if you write the tests after the code. In the next section, we'll talk more about what happens if you use test‑driven development.

Writing Tests After vs. Test Driven Development

As a developer, you're usually being paid to develop production code. As we've seen, developing a suite of automated tests is really useful, and I don't think it's an optional extra. Even if you do have a separate test team, they shouldn't be finding the kinds of trivial errors in your code that you could find yourself. You're also writing these tests for your team and for your future self, so you can more easily build on and adapt the code you've written for future situations. If you're not used to writing automated tests, you'll want to find a way to fit it into your personal development process. Most developers use what I would call a test after process. You write some code, perhaps do some manual testing in the console or the debugger until you're happy it's working, and then you write the tests. If the tests come after the code, then you don't invest in writing test cases until your design is relatively stable. This hopefully avoids rework. You know what needs testing and the interface to it should be stable. Having said that, this advantage may actually be something of a risk. You may find the design you've settled on is not very testable and only discover late in the day you need considerable rework before you can write the tests. You might also find bugs when you write these test cases, which you could have discovered much earlier. Unfortunately, this process can descend into a frustrating spiral of debugging and rework when you thought you were done. If you leave testing to the end, you might find you haven't got enough time to do a good job. Important test cases can be missed as you rush to meet a deadline. Test‑driven development is another way to fit developing tests into your personal way of working. I'd like to clarify I'm not talking about the wider development process behavior‑driven development or acceptance test‑driven development, which would involve additional artifacts and co ordination with people like business analysts, end users, and testers. Here, I'm describing a way of working that developers adopt while they're changing code, designing new functionality or updating existing functionality. It's the way of working that I prefer. Test‑driven development is a structured way of working where you interleave designing test cases with designing production code. Crucially, you allow the design of both the tests and the code to emerge as you develop rather than deciding everything about the design first and adding the tests afterwards. It's an iterative incremental approach to designing code and tests together. It's pretty unfamiliar to most developers though, and it takes a lot of discipline and skill if we're honest.

Demo: Test Driven Development

Let me show you how this works on the FizzBuzz carter. A code carter is a small exercise that people often use to practice their skills. So this is a description of the FizzBuzz carter. Write a program that prints the numbers from 1 to 100. But for multiples of three, print Fizz instead of the number. And for multiples of five, print Buzz. For numbers which are multiples of both three and five, print\_fizzbuzz. And then it helpfully gives you some sample outputs that your program should produce. So I'm going to start, of course, by creating a test case for this. And while I'm at it, I'll create the module where I'm going to put the production code, and I'm just going to arrange my windows so that I can see the problem description and my code on the screen at the same time. The next thing I'm going to do in test‑driven development is to think about the problem and how I'm going to approach it. And I do this by writing myself a short list of notes of the kinds of test cases I think I'm going to need to write. So I'm thinking that I'm going to need some kind of function that can convert 1 to 1, 2 to 2. But then if I give it 3, it should give me Fizz. If I give it 5, it should give me Buzz. And if I give it 15, then I should give it FizzBuzz. And that's kind of like the heart of this program that it should be able to make that kind of conversion. But the problem description wants me to print the numbers up to 100 as well, so I'm clearly going to need to be able to print up to 100 as one of the things I do. So that's just a little bit of analysis I do before I start writing any tests. A lot of programmers will just have this list in their head, but I like to write it down to help you understand what I'm doing. Okay, so let's write that first test case. So if I call this function FizzBuzz on one, I should get back 1. So there's that function. I've just decided it should be called FizzBuzz, and it doesn't exist yet, which is why it's highlighted red. But let's go and create that. And the parameter can just be called number. Okay, so this test now fails, good, and it's quite easy to get that to pass quickly. I'll just have it return 1. And that, of course, is not a good enough implementation. I'm not done yet. I've got a whole list of tests I want to do after this one. But it's a start, and it gives me that confidence that everything's green. I can refactor now because that's the next step in TDD. Write a failing test, make it pass, refactor. So now I can move that in the refactoring step into the production code and just check everything still works. Yeah, great. So I can cross that test off my list, and I can start on the next test. But I'm kind of thinking, actually, I don't think I need a whole new test case for this. I could just rename this one and have an extra case because actually they're very similar. I'm doing something called triangulation here. I'm just putting another example in to force myself to generalize the code. So if I just generalize the code like that, it will work for both those cases and probably any other cases I can think of. Okay, so let's try out three\_is\_fizz. So if I give this function 3, I should get back Fizz. And this test should fail now. Yep. So let's go and just fix that. The easiest way is just to put in an if statement, and I'm going to go straight for detecting a multiple of 3, not just the number 3 because my problem description says very clearly multiples of 3. So let's go straight for that, and then I can return Fizz. Okay, I'm feeling kind of confident that 3s and multiples of 3 are working. I could have put another assert in there for 6, but I'm feeling kind of confident. So I'm just going to go on to 5, and FizzBuzz of 5 should be Buzz. So let's see that fail as well. Yeah. And again, I'm going to go straight for the detecting multiples of 5 because I'm kind of confident that that's going to be what I need. So okay, I think 5 and multiples of 5s are working. So what about FizzBuzz when we've got a multiple of 15? So I think this should be fairly similar to what I've done so far. I'm building up a lot of confidence that my function is working with all these tests. Okay, so it clearly doesn't work for Fizzbuzz yet, but let's put in that code. There we go. Oh, didn't quite work. Made a typo. Good thing I had that test. Good. Okay, so I think my FizzBuzz function can handle the cases I can think of at the moment. So probably what I need now is to be able to actually print stuff. And rather than printing all the way up to 100, I'll just put an extra intermediate case on my list I think. Let's try something a bit easier first. Let's try just printing up to 3. That'll be easier to write a test for. So, I'm going to want to test for another function. Let's call that print\_fizzbuzz. And I'm going to assert that print\_fizzbuzz is going to, well, I should give it an argument, the highest number. And then I'm expecting some output. That looks like one to Fizz really. But actually, I'm not sure I want that to be the return value of this function. I think I want it to actually just print that to the console, which is what I'm trying to achieve. So let's just adjust this test. Let's make that a variable, which would be the captured standard output. And let's put in some code so that I can actually capture standard output. So that capsys there is a fixture in pytest that will allow me to capture whatever I print to standard output or standard error actually. So I can use it like this. First I call my function that's going print stuff. Then I ask capsys to read what it's got to stand out and standard error, and I'm just going to pick out standard out in this case, and then I can assert the contents of that. So I think that's what I want my new function to do. So let's go and create that function. And the parameter there is the up\_until\_number. Actually, highest\_number would be a better way to call that I think. And let's see. Of course, that fails. It doesn't produce anything to stand output yet. Okay, so I should get this to actually do something now. But before I get it printing anything, perhaps I just need to capture all of the FizzBuzz numbers up to the highest number. I can do that with a generator I think. And now I've got all the numbers. I should just be able to go through them and print each one. Oh, that didn't quite work yet. All right, yes, it's just putting an extra new line on the end. Good. Okay, so I think my function can print the numbers up to 3 now. But I'd better refactor, move that into the production code. And that seems to be working still. And if I just look at my list of tests, I think I'm just on to the last one now, being able to print all the way up to 100. And actually, instead of writing a unit test for that, let's just write the code and test it by hand. It should all be working actually. I should be able to just write print\_fizzbuzz up to 100 and run that. And yeah, that seems to work. That looks very much like the sample output that I was expecting. I'm getting some FizzBuzzes in there as well. Cool. So I think, actually, for this particular problem, I've solved the thing I set out to do. I've got to the end of my test list. And as always, when you get to the end of your test list, you think some more. Is there anything else that should have been on my test list? Are there edge cases I've forgotten? But actually for this exercise, I'm done, so I can actually just remove all of that. Notes that I made myself, I've turned them all into tests. I don't need the notes anymore. And I'll just rearrange my windows so you can see what I've done. Now, if I was unhappy with this design at all, I could still refactor this and rely on all my tests to catch any errors in my refactoring. So I'm feeling pretty confident in this code and this solution.

Test Driven Development Affects Your Design

I just demonstrated using test‑driven development to solve a small problem. A big benefit of TDD is the effect it has on your design. The first step in TDD is to think about what units you want to test. And we've known for many years that loosely coupled units with high cohesion is a sign of a better, more maintainable design. If you write the tests first, it will naturally push you in the direction of isolated units that are small because they are easier to test. Another aspect is the tests will exercise the interface to your unit. I mean they will call methods, specify parameters, and check return values. And this forces you to think first about the interface and second how to do the implementation. And that should lead you to have a better interface. TDD also gives you frequent opportunities to refactor. Every time the tests are green, which should be every few minutes, you can actually change the design, improve the code, move stuff around, increase the readability. It's an incredible freedom you have to change your mind and try out alternatives. Unfortunately, TDD is not a magic bullet that will make you a fantastic designer overnight, but it does push you in the right direction. I think it's interesting to consider what the design might have looked like for FizzBuzz if I hadn't used TDD. On the left is the design I did use TDD on, and on the right is an alternative implementation that I could have written if I wasn't using TDD. Both the code samples work. They meet the requirements. But the non‑TDD code is just one large unit, and the printing logic is mixed together with the logic to convert numbers. Of course, it's a very small piece of code, so it's hard to argue that this design is bad. But I think it gives you an idea of the kind of difference TDD makes, more smaller testable units. Now, I do test‑driven development, and I wouldn't like to work any other way, but I'm apparently not typical. This research study from 2015 used an IDE plugin to monitor developers as they worked. These were Java developers, but I have no reason to believe that Python developers are different in this respect. The researchers had over 13 person years worth of work to examine, and they found the majority of developers in our study does not test. Developers rarely run their tests in the IDE. Test‑driven development is not widely practiced. TDD is not an easy skill to learn. It's not something you'll become fluent with overnight. Learning about test design and doing this course will certainly help. There's more to it than that though. It's a way of working, a set of habits, and overall approach to development. If you want to learn TDD, I recommend practicing on code carters like FizzBuzz, pair programming with somebody who knows TDD, or you could get a skilled technical coach to come and work with your team and so you all learn it.

Continuous Integration

Whether you use test‑driven development or test after, the tests are a useful resource for all developers working on the code being tested. Continuous integration is a good strategy for collaboration and is enabled by having good tests. Tests are a shared resource for everyone working on the same code base. When you need to change some unfamiliar code and you find it has some tests, this should be an advantage. They will help you to understand what the code does. They act as documentation and to see if your new code breaks anything that used to work, regression protection. You should share your tests together with the code. And the usual way to do this is to put the tests in version control together with the code. Version control helps you to keep your work in sync with other developers. You'll want to store your test code in the same version control repository as the code it tests. Then when you come to do some work, first, you pull the latest changes from version control, so you get all your colleagues work. You run the tests, check they pass. And then when you've done your new development or whatever changes you need to make, again, you run the tests before you share your changes and push them to version control. What often happens though is people make mistakes, and they forget to run the tests, and you find unexpectedly broken tests and code in your shared version controlled repository. If nobody notices or acts to fix it, there's a danger that the tests will become worse than useless. If they're constantly broken or provide an out‑of‑date picture of what the code does, you'd probably be better off without them. That's why you add a build automation server to catch mistakes and help you keep your tests in a passing state. The build automation server listens for changes in your team's version control repository. And when it finds some, it gets the latest code and runs all the automated tests. If there are test failures, it will alert the team that there's a problem, send out emails, turn on a lamp, update a web page. If everything passes though, the server may deploy to a manual testing environment and alert testers there's new code to test. A build automation server is a huge help for keeping your shared tests passing. But just having one won't automatically mean you're doing continuous integration. In my experience, an effective and productive team will be making frequent, small commits and sharing them often. A change that one person makes will quickly propagate to all the other team members. You all have essentially a shared view of the same code that you're all working on. If everyone in the team shares their changes at least daily, then you can call what you're doing continuous integration. It's a strategy for collaboration, not primarily a tool or technology. But it does work better if you have a build automation server and automated tests. What you're trying to achieve is a situation where the differences between the tests and code on my machine and those on my other team members' machines is only a few hours work. The other extreme where you share your changes after days or weeks will often cause painful code mergers and badly broken tests. If you'd like to find out more about continuous integration, I recommend Martin Fowler's article on this topic.

Module Summary

In this module, we've discussed the benefits you can get from unit testing and the impact writing them can have on the quality of your code. I recommend using test‑driven development where you interleave designing the tests and the production code. I also recommend continuous integration where everyone on the team shares their changes often, and unit tests are run frequently. However you do it though, the important thing is to be sure to write automated tests alongside your code, share them with the other developers who also work on that code, and make sure they keep passing.

Using Test Doubles

Module introduction

In this module, we will learn about using test doubles to replace awkward dependencies. There are several kinds of test doubles, and we will go through them all, stubs, fakes, dummies, spies, and mocks. You'll learn about when to use each kind. You should use a test double when the unit you want to test has a dependency on some other code and for some reason you don't want to use the real one. You can replace that code with a test double. A test double is like a stunt double. That's where the name comes from. In a Hollywood movie, you might want to have a scene where your actors do something rather dangerous and dramatic, swing through the air, battle a monster. Rather than risk your expensive starring actor, you'll probably hire a stunt double who will do all the dangerous stuff. The trick is to make the people watching the movie believe that it's the actual actor. So they dress like them and they wear a wig. They impersonate the real star. It's similar with a test double. It's standing in for the actual class, the one that's used by the thing you're trying to test. And the class you're testing shouldn't realize it isn't talking to the real collaborator object. They look the same. But of course, underneath the test double is different. It's completely configurable so you can control what happens to your class while it's being tested.

Demo: Stub

Gerard Meszaros invented the term test double in his book xUnit Test Patterns, and he meant it as an umbrella term for all the various different sorts. The trouble is most people still use the term mock to mean all kinds of test double. I think it's useful to realize that there are different kinds, and the term mock actually also means something much more specific. It can help you to know which thought you're using and in what situations it would be suitable. I'm going to go through each sort beginning with the stub. Let's look at a demo using a stub. Imagine you're writing some code for a racing car computer system, and you need to alert the driver when the tire pressure falls outside of an allowed range. You have an Alarm class in your system, and it uses a sensor to detect the pressure in the tires. If the pressure is too low or too high, it switches the alarm on so the driver knows they should take a pitstop and sort it out. This is the code for the Alarm class. It has a constructor that initializes the high and low pressure thresholds. It holds a reference to the sensor and has a property, is\_alarm\_on, with the alarm initially off. In the check method, we ask the sensor for the current pressure. And if it's outside the allowed range, we switch the alarm on. The trouble is the driver has reported a bug. The alarm is not being turned on when the pressure is exactly 17.0 or 21.0, and it should be. We'd like to write a test that fails because of this bug. The trouble is the tire pressure sensor is awkward to use in a unit test. It makes a call across a network to an actual racing car with tires and a physical sensor. We want to test the check method, but it has this awkward collaborator, sensor. It would be more convenient if we can substitute this class with a test double. There's a slight problem here. The Alarm class constructor uses the sensor constructor directly, which makes it a little bit difficult to substitute it for a test double in the test, but have normal clients continue to use the real one. We already have a test for the constructor that passes, so it should be safe to refactor. I'll make the sensor an optional argument so the test can pass in the stub, and the normal case will continue to use the real one. Now I can work on a new test case. The alarm should be on at the lower threshold. So the basic structure of the test is I want to construct an alarm, call the check method, and assert that it's on. But of course, I need to know that the alarm should be on because the sensor is giving the lower threshold. So here I can pass in my stub sensor to the constructor, and I need it to have the same method as the real sensor for sampling the pressure. But I'm going to tell it to return the hardcoded value 17, which is the lower threshold of the alarm. And this test actually fails to expose the bug. It says the alarm should be on, but it's not. And then if I go and actually fix that bug, I should be able to see the test passing. Yes. So that means that I was able to use the test double, the stub sensor, to replace the real sensor, and then I know what the pressure is and that the alarm should be on. I'm going to do another test for the higher threshold, which also has a bug. But I'm going to use a different kind of test double, well, the same kind of test double, but using a different way to construct it. I'm going to use this built‑in mock that comes with the Python standard library, and it's in the unittest module actually. And you pass to the mock constructor the class that you want to mock so that it will know what methods that class has that you might want to actually substitute. And then you have to configure it to say, well, on the sample pressure method, I want the return value to always be 21.0. That's the higher threshold. Those two lines of code do the same thing as the stub sensor class I declared before. Don't be confused by the name mock here. As I mentioned, most people say mock when they mean any kind of test double. Here I'm using this mock function to create a stub. As before, the test fails and exposes the bug. The alarm is not on when it should be. I'm just going to rearrange my windows a little so I can see the code better, and I'm going to fix that bug there. And hopefully now my test should pass. Great. So here you can see I've got two tests using a stub, and you can see that they are both able to force the code to go through a certain path and expose a bug. And that would have been difficult to do in another way. In the demo, we used a stub sensor to replace the real sensor. The stub has a very simple implementation with no logic or behavior, and it lets the test control exactly what the collaborator returns to the class we're testing. Like every test double, the characteristics of a stub are that they look the same as the real class from the outside. They have the same methods. However, what is inside is pretty much nothing. Each test you set up, you have to give it a hardcoded response to the methods that the class and the test is using. Out of all the kinds of test doubles, stub is almost the simplest. You use one when it's inconvenient to use the real collaborator object. The real object could be unreliable, difficult to construct, slow, access resources that are unavailable in the unit test, and the stub is a very simple replacement that you control.

Demo: Fake

We've just talked about stubs. Now I'd like to discuss how to use a fake. A fake is rather like a stub actually. Let's look at an example. Imagine you're writing some code that will convert some text from a file into HTML. You have a class HTMLPagesConverter, which has a method, get\_html\_page, that will open a file, find the relevant part of it, and convert that to HTML. So this is the method you want to test, get\_html\_page, and the file is the collaborator. It's a slightly awkward collaborator in a unit test because accessing the file system is relatively slow, so I would prefer to replace it with a test double. We could use a stub. The thing is, that would be a little bit tricky. We're using several file operations, seek, tell, detecting the end of the file. We could stub those methods, but it starts to get complicated. In this case, I think it would be easier to use a fake. In the Python standard library, there's a class called StringIO. It has all the same methods as a file, but it's entirely in memory. It might not be possible to use a StringIO in the real production code since the files could be very large and they need to persist between runs of the program, but we can use it in our unit tests as a fake to replace the file. So a fake has an implementation with logic and behavior, but it's not suitable for production for some reason. Here's the source code for the HTMLPagesConverter. In the constructor, we are given an open file and then we scan through it to find the page breaks. The method that we want to test is get\_html\_page. You ask it for a particular page from the file, and it will go and retrieve it. It will also do some conversion of the text it finds in the file and adds HTML tags. Unfortunately, there is a bug in this code. I've marked it with a comment. It's going to retrieve more pages than we asked for. Let's write a test to reveal this problem. I want to write a test case where there are three pages of data, and I want to fetch the second page. So I'll construct a pages converter. I'll come back to the constructor in a minute. And what I need to do then is to ask it for the page, the second page of the three. And then I can assert that it contains the text I was expecting on page 2. And of course, it will add some HTML tags to that. That's what this code does. Now, I want to pass it that open file, and this is where the fake comes in. Instead of using a real file, I'm going to construct a StringIO, and that takes to the constructor the string that should be in the StringIO that will pretend to be the file contents. So I need to add three pages of data for this test with the correct page break marker in between. So hopefully, my HTMLPagesConverter will be none the wiser. It will think it's got a real file with three pages in when actually it's just got a string wrapped in a StringIO. So when I run this test, it's actually failed, which is good. It's exposed the bug. There it's expecting just to get back the text for page 2, but it's getting page 3 as well. And that's the bug that we found. So now if I go and actually fix that bug and so I only get one page, the test will now pass. And we can just review the code. The test case is passing this fake file contents to the HTMLPagesConverter, and then it can test the behavior. The characteristics of a fake are that like every test double, they look the same as the thing they're replacing from the outside. They have the same methods. But unlike a stub, they do have a realistic implementation inside. The implementation is in some way unsuitable to use in production, but it's similar enough for test purposes. In this example, we replaced a file with a string. In a similar way, you could replace a database with a fast in‑memory database or a real web server with a lightweight one. These kinds of fakes usually already exist and are used in production in other contexts. Often, you use these as fakes to make your tests run faster and so you don't have to write as much code for stubs.

Demo: Dummy

Next up, let's talk about dummy objects. We'll look at an example. I've got a function here called fizzbuzz that can convert a number to its name in the game FizzBuzz. I also have some tests for it, and these tests pass. I have an idea for a new feature. I'd like to be able to configure it with additional rules. So I could have 7 be Wizz, and then 21 would be FizzWizz, and 35 would be BuzzWizz. I like the sound of that. I'll write a new test case for the new feature. The first test case will be checking that 7 is Wizz if I pass these additional rules. So I need to just change this test case so that those additional rules will be passed to it, and then I can pass them to the function under test. Of course, this test fails at first because it's not using the additional rules. So let's just add the code that does that. If I'm given additional rules, we should add them to our dictionary. Ah. Okay, so the new test is passing. But, however, the existing tests are now failing. And it says, ah, of course, I need to pass the additional argument. Since these tests don't actually use the additional rules, I can pass none. That solves the problem. And actually, that's why I wanted to show you this. This is an example of a dummy. In place of a collaborator, additional rules, we're providing an alternative actor, just like with the other kinds of test doubles we've looked at. And in this case, the alternative is just none. A dummy is usually just none or sometimes an empty collection. The thing is, it's actually a design smell telling you the method you're calling has too many compulsory arguments. I'll change that and add a default value instead. That's better. Now my first test case no longer needs a dummy, and my design is clearer. So that was a dummy. It doesn't matter what it looks like. It's not used. You need a dummy when you're forced to pass a collaborator to the method or class under test, but that collaborator isn't used in the scenario you're testing. It's actually arguable whether a dummy is a test double at all since it's often not even an object. It's usually none.

Kinds of Test Double - Summary of What We've Learnt so Far

Let's take a moment to compare the kinds of test double we've looked at so far. Dummies are usually none. Stubs are configured to respond with fixed preprepared responses where fakes have an actual implementation that works. You turn to one of these kinds of test double when it's not convenient to use the real collaborator in your test. As you move from dummy to stub to fake, they get more similar to the collaborator being replaced. I've talked about collaborator classes here, but note this could equally well apply to a more functional design. You can use a test double to test a function that takes another function as an argument. You stub or fake that collaborator function. The remaining kinds of test double are spies and mocks. Mocks and spies actually do everything that stubs do. They have the same methods as the object they replace, and they also return hardcoded values and have little of their own behavior. The big difference is unlike stubs, they make assertions about what happens in the test case. A mock or a spy can cause a test to fail. A stub won't. There are actually three kinds of assertion. Return value, you call the function you're testing and check the return value or expected exception. The second kind is a state change. You call the function you're testing and check the side effects happened, something changed the state. Those two kinds of assertion, checking your return value, checking state, are probably the most commonly used in any kind of test, and they're the only ones we've used so far in this course. The third type of assertion is where you check a particular function or method got called. Now this is a more complex kind of assertion than the other two, and it uses a mock or a spy, which is what we'll look at next.

Demo: Spy

Let's look at an example where we want to use this third kind of assertion using a spy. I have a DiscountManager class that adds discounts to products. It has a collaborator class, Notifier. When a discount is created, we want to notify any users who recently bought this product. We already have a test case for this that uses a StubNotifier. The stub doesn't send any messages to any users, which is quite convenient for a unit test. Unfortunately, we have a bug being reported from production. Some of our users are getting a lot of messages about discounts, but most users are getting no messages at all. We think the message sending functionality in general is working, but we suspect there is a bug in the DiscountManager. This is the code for the DiscountManager. It has a method, create\_discount, that should add the discount to the product and then notify users. There is a bug here that I've marked with a comment. We're only sending messages to the single key user rather than all the users that previously bought the product. We actually already have a test for this method. It checks that the product gets the new discount. The notifier is an awkward collaborator. It sends emails. So this test replaces it with a StubNotifier that doesn't do anything. The thing is, to expose this bug, we'd like to write a test that will let us check the correct emails are sent. I'm going to change the StubNotifier into a SpyNotifier that will be able to do that. When it gets a call to notify, I'm going to record that in a list. I'll just have to create that list now in the constructor of the spy and improve the documentation. So now my spy will be able to tell the test which users got notified, and I can assert on that. It should be both user number 1 in the list and number 2 in the list. So this test now fails because of the bug, which it didn't before. It fails because only the key user, which is user 1, is being notified. And to be honest, it's not very easy to read the error message. I think pytest gives us information overload. But if you look at the line that actually failed, it's the second user that's not getting the notification. So I'll fix that bug. And now both users are notified, and the test passes. I'm actually going to put the bug back now because I want to show you a second way to do the spy, this time with a mocking framework. So I'm going to just copy that test and change the way it's named so that it's clear that this one is the same, but with a mocking framework. And that one is with the spy that I handcoded. I'm using the same mocking framework as we used before, the one that's included in the unittest module. Then I can change the way I do the assertion. This mock or spy has got a method assert\_has\_calls, which takes a list of expected calls. Each of those call objects should specify which arguments this call should receive. So, the notify method has two arguments, which user should be notified and the exact message that they should be sent. So I'm just going to copy that from the production code so that I get it right. So I'm expecting two calls, one for each user, but actually with the same message. Okay, so now both my tests are failing because of the bug, and this is the one with the mocking framework. I'm not actually convinced that this error message is any clearer than the last one, even though pytest has given us loads of information and quotes the documentation. Here at the end, it does actually say that the actual calls were the two calls to user 1 instead of one to user 1 and user 2, but it's a bit hard to make out. I'll fix the bug anyway, and we can see that both tests pass. We replaced the stub with a SpyNotifier that keeps a record of the calls it receives while we're using DiscountManager in the test. And then afterwards, we can ask the spy if it received the calls we expected and, if not, fail the test. So like every test double, spies look the same as the real class on the outside. They have the same methods. And otherwise, they're rather like a stub or a fake. You may have hardcoded return values or a real implementation. The key difference is the spy records what happens to it. It spies on things, so you can make assertions about what happened afterwards.

Demo: Mock

We've been through all the types of test double now except for mock. So let's look at that one. A lot of people say mock when they mean test doubles in general, but there is a more specific definition. Let's look at an example. I'm going to continue with the previous demo, the DiscountManager, and I'm going to start by reintroducing the bug so that I can write a third test this time using a mock. It's going to be kind of similar to the one that uses a spy, so I'll copy that one. So this one is going to use a MockNotifier. So let's create that class. And to start with, I'm going to just make this work as it did originally so that the mock is like a stub. And to act like a stub, it needs to have the same interface as the class it's replacing, but not do anything. Okay, so that test passes. It doesn't expose the bug, but it does actually run. So now, this is the part that makes this a mock and not a stub. I need to set some expectations on the mock for the calls that it should receive in the next part of the test. So it should expect a notification to both users. And if I add the implementation to my mock, it's going to keep a list of the expected users that will be notified. And I'll need to add that field and initialize it to an empty list. Okay, so now in the notify method, I can actually check that I get the calls that I'm expecting. I'll look in my list, which is the expected user that I should get now. It should be the one at the front of the list. And then if the user I actually got wasn't the expected one, I can raise a runtime error with an informative message. I got a notification for the unexpected user, this user, and I was expecting this other user. So that is a mock. You set it up in advance with expectations for the calls that should happen. And then if they don't happen as expected, you raise an error. When I run this test, now it fails because the mock is not having its expectations met because of the bug. And you can see that the failure line in the test is the act step. It's the call to create discounts where the test has stopped. It stops because a runtime error is thrown in the mock. The message shows that we received a notification to user 1 when we were expecting user 2, although it's still a bit of information overload there. I think I could just improve that. So now with the names instead of the whole user objects, it's actually much clearer. So then if I go and fix the bug, of course, this test will pass just like the others. This time, we replaced the notifier with a MockNotifier. We set it up to expect particular calls. And then, when we call DiscountManager in the test, it will use the MockNotifier. If the mock receives an unexpected call with the wrong argument, it will throw an exception and fail the test straight away. So like every test double, mocks look the same as the real class on the outside. They have the same methods. Otherwise, they're very like a spy. The main difference is that the mock is told in advance about what methods will be called with what arguments. And if those expectations aren't met, it will raise an error straight away in the act part of the test. The spy raises an error afterwards in the assert part of the test.

Comparing Spies with Mocks

A spy and a mock are very similar. Both check for correct interactions and can fail the test. The main difference is the mock will explode earlier in the test, and that can be helpful if you want to stop execution as soon as something goes wrong. You get a stack trace that points right to the heart of your production code. A spy will only tell you afterwards what happened. Having said that, spies are normally simpler to implement, easier to understand, and make your test code more straightforward. Test doubles form two groups and are used in different situations. If you have an awkward collaborator that is difficult to use in a unit test, replace it with a dummy, stub, or fake. Most of the time, a stub is a good choice. Spies and mocks are different in that they can fail the test if the unit under test doesn't interact with them correctly. Use them to check a contract between the unit under test and the collaborator. Check that particular method calls happened in the right order or with the correct arguments. Different people have different styles of design. Some will use mocks or spies quite a lot, relying on them to help design how classes interact. Others use them hardly at all.

Module Summary

A test double replaces a collaborator of the class or method or function that you're testing. There are different kinds of test double, stubs, fakes, dummies, spies, and mocks. You usually use a test double when the real collaborator object is awkward to use in the test. It could be slow or difficult to construct. I often see people using test doubles to test poorly designed code and it making things worse. We'll talk more about that in the upcoming module, code that's hard to test.

Code That’s Difficult to Test

Module Introduction

In this final module of the course, we'll discuss how to handle code that is difficult to test. In particular, we'll look at how to make hard‑to‑test code into easy‑to‑test code using strategies like peel and slice, monkey patching, and self‑initializing fakes. The techniques in this module are for handling code that's hard to test. This kind of code generally has poor separation of concerns, lack of encapsulation, and generally poor structure. You won't particularly need these techniques if your code is well‑designed. All too often though, I see code that needs refactoring. And before you do that, you want to have some tests in place as a safety net. This module is about the techniques you need to get those tests in place. So let's talk about the opposite, code that's easy to test. It's usually some kind of pure function, that is logic without side effects or external dependencies. Most of the example code in this course has been fairly easy to test because it's been mostly pure functions. The section on test doubles was an exception. There we had example code with dependencies, but they were relatively easy to isolate with a test double. Effectively, the parts that made the code not pure were the parts we replaced with the test double, and the remaining code was much more pure, much easier to test.

Demo: "Peel" Strategy for Hard-to-test Code

The first strategies I'm going to talk about are for when you have code that is mostly testable pure logic, but with a small amount of difficult‑to‑test code mixed in. The strategies are peel where you take away the untestable outer part like you peel a banana and slice where you take away the untestable inner part like you slice away the stone in a cherry. So first, the peel strategy. If all the hard‑to‑test code is at the start and the end of the method, you can peel it away to get at the juicy easy‑to‑test middle part. You do extract method and create a new testable method with more arguments than before. The original function now contains very little code, and it's all about side effects and awkward dependencies. That part's hard to test. But the reason you do this is because the new method should be east to test, and that is a great improvement on the original situation where none of the code could be tested. Let's look at an example using the peel strategy. This is the code I'd like to write some tests for. It's a fragment of a larger system for forecasting sales of ice cream. This get\_score method has some juicy logic that I'd like to write tests for. The first line here makes a call to lookup\_weather, which here is just a placeholder implementation. The real implementation would make an API call to some kind of weather service. Here I just have it returning a random value to simulate that. But that is going to make this method difficult to test. Another problem here is the use of this global variable, flavour. I started to write a test for this code. I worked out that if no flavour is set, the score should be ‑1. If I run this test by itself, it passes. Unfortunately, if I run it together with all the other tests, it doesn't. One of the other tests that's failing there is also changing the flavour that's set. I'd like to use the peel strategy on this method. I'm going to select this large part of the method which I think should be testable, and I'm going to extract it as a method, get\_score\_with\_weather. And as you can see, it has an argument now, whether it's sunny or not, which is the return value from lookup\_weather. So that means this method should be much more testable. If I say that the weather should be sunny and I set the global variable flavour beforehand, I should be able to work out what I expect the score to be. So that looks a little bit more hopeful. I would like to deal with this global variable in a better way though. I'm going to add it as another argument to this method. Now the parameter to the method is shadowing the global variable. I can rename it, and that makes it clear. It's not using the global variable anymore. I need to update my test to pass this value as a variable now. So this method is pretty well testable now. It's no longer using the global variable or the awkward lookup\_weather. The original get\_score method is much smaller, and it's still hard to test though. But since there is so little logic there now, we might be satisfied with just testing the other method. The coverage that we have isn't so great though. Let's add some more tests for this method. I'd like to use the same approach that we used in the last module with combination approvals. This function is eminently testable. I'm going to need a list of all the different kinds of weathers, both sunny and not sunny and the different flavors. I'll also need a printer. The simplest thing is just to print out the arguments and what the result was. Of course, it fails the first time I run it. But this received file looks great, so I'm going to approve that. And when I run it with coverage again, I can see the coverage is much better than before, even if it's not perfect. So, I feel pretty confident I've got the main interesting logic in this function under test now.

Demo: "Slice" Strategy for Hard-to-test Code

The second strategy of the pair is slice when the hard‑to‑test part is in the middle of a lot of juicy testable logic. So it looks like this. The hard‑to‑test code is a small piece in the middle of the method. Peel doesn't work when the code is like this. What we do is create a local function that does the hard‑to‑test code, and you move the function declaration to the start of the method and call it from the original place in the code. Now, you've got a situation where you can peel that logic. Let's look at an example. This is a different part of the ice cream forecast sales application. Print\_sales\_forecast is a function that also has testability challenges. A couple of things are obvious. It uses the current date and time, a dependency on a global variable, and it prints to standard output, again, a side effect at the global level. These things we know how to fix though. This is the pytest test fixture capsys that we saw before. It allows you to spy on the contents of standard output. So this test is already looking quite promising. We can see the report. Handling the current date and time is my next project. I'm going to copy this value so I can use it as a fixed value in my test. I'd like to be able to pass in the current date and time with this keyword argument now. I'll add that here with none as the default. This is a standard pattern that I often use if the code depends on a global variable like this. So this code looks like it should be a bit more testable now. I'm going to go over to using approval tests. I think it's a bit easier to use than pasting those kind of values into the test code. I'd rather have it in a separate file. So I've approved that result, and now I have a test that apparently passes until it doesn't. It turns out there was another testability problem in this code that is now obvious. This line of code here calling scorer.update\_selection actually has global side effects. This is the implementation. I've put in a random choice here as a placeholder implementation for a real system that would use some kind of machine learning subsystem to predict sales. It suffices to show that I can't call this method from my unit test. Looking again at this method, the line that it is hard to test is right in the middle of the logic. Here, I can't do a straightforward peel. I'm going to need to use slice. I'm going to create a new local function called update\_selection that calls this line of code, and then I can replace it here with a call to my new local function. Now this should be a pure refactoring. My test still passes sometimes and fails sometimes. I haven't actually solved the problem. But this refactoring that should be relatively safe does allow me to now do the peel strategy. I'm going to peel this method and name it print\_sales\_forecasts\_with\_update selection. And one of the arguments to the new method is a reference to that function, and that will enable me to replace it with a test double from my test. I'm going to update the test case now to call this new testable function. And I'm going to need a stub for that update\_selection function. I'll write one by hand here. Checking what the real one does, it's going to update the global flavour variable in the scorer module, so I'm going to set that to a fixed known value so that the behavior is no longer random and unpredictable. So now when I run this, it does fail because the result is different this time. But now it should be consistently this new value. So I'll approve that. And when I run it, it now passes consistently. So this is my updated production code with the use of the slice strategy. I'll run my test one more time, this time with coverage, so you can see clearly one of the disadvantages of the peel and slice approaches. The new testable function is pretty well covered, but the original function, although much shorter now, is not covered at all, and all the logic that is left there is awkward and difficult to test.

Consequences of the "Peel" and "Slice" Strategies

The peel and slice strategies are for code that is mostly testable pure logic, but with a small amount of difficult‑to‑test code mixed in. You separate out the logic and get it under test, leaving the remaining code not tested. When you've done this, the overall design may be worse. This is a short‑term strategy to get enough tests in place that you can begin refactoring and fix that design. We also need a strategy for code that is mostly side effects and almost no pure logic. You can get this kind of code because you peeled away all the juicy testable logic. But also, some systems are just like this. They have a lot of integration code. One strategy is just to not have automated tests for that part specifically and rely on manual tests. It needs to be really obvious if it breaks though so you notice quickly, for example the API code used by your front end. If you have people doing manual exploratory testing via the GUI, they should spot if the connection to the back end breaks. If the code is very standard following an accepted design pattern, you might be satisfied with extra careful code inspection and just a few manual checks, for example if the code does MapReduce or another standard pattern for concurrency. If it really needs automated tests, you generally use a test double instead of the thing you're integrating with. We talked about this in the test doubles module. The thing is, it can be quite difficult to do this well. There are many pitfalls. Let's look at some more complex examples.

Demo: Monkeypatching as a Way to Insert Test Doubles

Monkey patching is another name for metaprogramming. It's when you dynamically change an attribute or a piece of code at runtime. You exchange the code that was there when the program started running with some other code. And this can actually be a very useful way to insert a test double. Let's look at an example. This is the code as we left it at the end of the last demo in scorer. We used peel to separate out the lookup\_weather from the rest of the logic. Before, we had a placeholder implementation for lookup\_weather. I've now replaced that with a proper implementation that uses the requests module to get a weather forecast from an external service. The tests I wrote before give me good coverage on my logic, but, of course, no coverage at all on this lookup\_weather method. They also don't cover the get\_score method that I began with. I'm going to write another test for this lookup\_weather function. I'll start with a test for the default location. I'm going to assert that the weather, if I look it up, is going to be sunny. It's going return true. And I'll need to fix the rest of the test so that that actually is the case. To start with though, I'm just going to use the debugger to step through this test as it accesses the real service. So there I got a response from the remote service. It says 200. So then I go on to check out the JSON that I get from that. And in the JSON, I can see that under the weather main key, I get sunny, which is convenient because it means it's going to return true from this function, which is what my test was expecting. Okay, so that's all very well, but I'd quite like to have a test that I could run when the remote service is not up and running or when the weather isn't sunny. So I'm going to replace that lookup with a remote service with a test double, and I'm going to use the monkey patch test fixture from pytest. It has this method, setattr, which allows me to replace with a monkey patch a function in the requests module with a stub or another kind of test double. So here I've told it that I want to replace the get method in requests with this stub function that I'm just writing here. And let's have it return a StubWeatherServiceResponse. I'll create that class, and let's see what it needs to have. So the request.get method returns a response, and the response has a status code. So let's add that to it. Start with 200, and it also has JSON. And for now, I'm going to get it to return just the bare minimum amount of JSON that my code is looking for. It's looking for a key weather and then a key main, and we'd like it to return sunny. So the test still passes. It's hard to see that there's a difference. So I'm going to step through it again in the debugger and show you that this time it really isn't accessing the real service. It's using my new stub. So there we've got the response, which is a StubWeatherServiceResponse rather than a real one. And when we call json on that, we get back a very small piece of JSON with just enough that it can determine that the weather is sunny. So I'm running this again with coverage, and I can see that the coverage on the lookup\_weather method is much improved now. So this test code is actually a reasonable example of using monkey patching to replace the requests module so that we can become independent of an external service by using a stub. I'd like to show you another example, which isn't one I would recommend. If you remember this method get\_score that we started with has no tests. It's only two lines long, but I can get this under test by using monkey patching. I'll need to call the get\_score method. It doesn't take any parameters. And that's one of the reasons I need so many monkey patches. The first thing it does is it calls lookup\_weather, so I'll need to patch that. I can replace that with a stub. I'll have it return that the weather is sunny. Next, I call get\_score\_with\_weather, and I use this global variable flavour so I need to patch that, and I'll set a value of Vanilla. And then, this method get\_score\_with\_weather is already being tested. So we don't need to test that one again actually. So I can use a mock just to check that it's called with the right arguments. I'll define my mock here and check that it's getting the right arguments. So sunny\_today should be true, and the flavour should be Vanilla. And I'll just return 0 and just check that I get that. I'll run this test with coverage, and I can see now that those two lines of code that were not covered before are being covered. But at what cost? This is not the kind of test case that I'd like you to write. Let me show you how fragile this is. If I just rename this method, then, of course, my test no longer works because my mock isn't being introduced. I'm replacing the get\_score\_with\_weather method, not the get\_score\_with\_weather\_and\_flavour method. This is a very bad way to use monkey patching. I'm using it to replace details of the scorer module, which is the module I'm testing, details that might well change in the future like method names. And there is so much code here setting up mocks and stubs compared with the size of the code under test. I really don't think it's worth it. Previously, I've shown you how to replace collaborators with test doubles by passing them to the constructor of the class under test. That's like using the front door, the public interface of the class under test. Sometimes though that's inconvenient. Perhaps the collaborator is a global module like requests. From the test point of view, it's like it's behind a locked door. Monkey patching is a way of getting a back door so you can insert a different collaborator. You should realize though that test doubles always come with a cost, and it's often more so if you introduce them with monkey patching. First, they add complexity. Test doubles make your tests harder to read and understand than they would be without them. You can end up not testing what you think you're testing. Test doubles get out of sync with the real object quite easily, and you can find your test is passing, but the real object has a changed behavior that your mock doesn't do, and your code actually no longer works. Your test double can hide that problem from you. Test doubles can also hinder refactoring. We saw that here with just a simple thing like renaming a method, but it can be worse if you also create new objects and move methods around. Your tests are tied to the details of your implementation. Monkey patching can make this worse because you can replace parts that are not meant to be part of the interface and can be very volatile.

Demo: Self-initializing Fake

A self‑initializing fake is another kind of test double that you can introduce via monkey patching. It can reduce the amount of test double code you need to write. Let's look at an example. This is the test I wrote before for the lookup\_weather function using a stub weather service that I monkey patched. Let me show you another way to do this. I'm going to write another test case for the case when the weather is not sunny. And for that, I'm going to need a different location, and I happen to have one which I've prepared. This is the coordinates of Are, which is a Swedish ski resort, and we're going to look up the weather there, and I'm guessing it's not sunny. I'm going to run this in debug mode so you can see the response I get from the actual server. So here we can see the JSON we get back from the server. The main weather in Are is snow, which is very nice if you're skiing, but not so much if you're trying to eat ice cream. So, we would like to make this test be reliable even when the server is not available or the weather in Are is sunny, which it is sometimes. So I'm going to add a few more dependencies to my project. I'm going to add vcrpy, the Python version of VCR, and pytest‑recording, which is a plugin for pytest that plays nicely with the vcr package. And then I found that I had to fix the version of urllib3 that I was using. I expect by the time you watch this, they might have fixed it. But vcrpy seems to need the slightly older version of urllib. So when I've installed those things, then I can add an annotation to my test case, mark.vcr. And I'm going to run the test with a new flag. I have to tell vcr how I want it to do its recording. And the record mode that I prefer is once, which means that it will record any new interactions the first time. And after that, it will expect to be able to replay them. So this is the first time I've run it with the annotation, and that means it should record for me what the server has done. So here, I've got this new file that's appeared in this folder cassettes. And it's kept in a directory the same as the name of the test file, and the name of the new YAML file here is the name of the actual test case. And this contains the details of the interactions that I made using the requests module. And it's showing all of the headers and everything and the URL that I was accessing and the body of the response, the JSON. So you can see there the main weather forecast is snow, and it's recorded all of that for me. If I run this again, it will pass again, which is perhaps not so impressive because it passed last time. And let me just show you the difference it makes if I have another test that's exactly the same, only doesn't have the vcr annotation. So if I run this, they still both pass because the server is still up and running, and the weather in Are is still snowy. But if I go away and actually stop the server and then if I run them both again, you can see the difference that the test that's not using vcr is now failing because the server is not there; whereas, the one with the vcr recording is still passing exactly the same as before. And that's because it's using this file, this cassette that it prepared earlier to provide the response from the server. So if I put all this on the screen together, you can see the difference. With the vcr package, I can write much less code. If at some point in the future, the response from the server changes and I need to update this test, I can just delete my cassette file and rerecord it from the server. The vcr package helps you to create and maintain what Martin Fowler has named a self‑initializing fake. It works like this. If the system under test needs a remote service, in your test, you'll need to replace that with a test double, a fake. The first time you call this self‑initializing fake, it forwards your call to the remote service and records what happens in a file. Then the next time you call it, it just provides the response from the file. The remote service doesn't even need to be running. This approach still has pitfalls. It's still using monkey patching underneath to replace the requests module. However, that API is really very stable, so it's not a big problem. I think the test code is actually more readable than before. The annotation is fairly non‑intrusive. Cassette files, the recordings of the interactions, might get out of date compared with what the real service does. That's the biggest danger I think, although it is relatively easy to rerecord the cassettes. You just delete the file and run the test again as long as you still have access to the real service and can easily set it up to give the responses similar to before. As with any test double, this one can also hinder refactoring, particularly if you rename tests since the cassette file is named after the test. Otherwise, I find it's easier to refactor this kind of test code than with handcoded test doubles.

Module Summary

This module was about code that is difficult to test. In particular, we talked about how to make code more testable using the peel and slice strategies and, if the code has dependencies that are awkward to replace by normal means, how to use monkey patching and self‑initializing fakes to get it under test.

Improving Test Coverage and Maintainability

Module Introduction

In this module, we'll talk about improving test coverage and maintainability. We're going to introduce approval testing, which is an alternative to using assertions, code coverage measurements, which can help you to improve your tests, all combinations, which will help you to achieve better code coverage, and mutation testing, which can help you to evaluate how good your tests actually are.

Demo: Replacing an Assertion with Approvals

I'd like to begin showing you approval testing with a demo. We'll replace some assertions with approvals. This example is some software for a supermarket, in particular the teller machine that cashiers use to calculate the price of a shopping cart. We have a catalog of products. We're using a test double here, a fake, with two products, a toothbrush and some apples. Today, we also have a special offer, a 10% discount on toothbrushes. In this test, we're going to put some apples in our shopping cart and ask the teller for a receipt detailing our purchase. That's the act step of the test. After that, we have several assertions, checking the receipt is correct and that it contains one receipt item for the apples. This test is passing, but I would like to replace all of those assertions with approvals. First, I need to actually install approvaltests in my project. I'm also going to install pytest‑approvaltests, which makes it more convenient to use approvals with pytest. Okay, back to the code. Now I can work on my test case and use approvaltests.verify. So I'll import approval tests. And here, the argument to verify is a string. I mean it can take other things as well, but the simplest way to use it is with a string. So I need some way of turning my receipt into a string suitable to use here. Now I prepared earlier a class which is called ReceiptPrinter, which has this method, print\_receipt. And this code is not all that complicated. You can see it here. It takes the receipt object and goes through all the items in the receipt and prints them to a string, and then it goes through all the discounts and prints them and adds them to the string and then presents the total of the receipt and basically just returns the string. So this is just turning a receipt object into a nicely presented string. So, now we can actually run this test again. If you remember, this test passed the last time we ran it, and now it's failing. Now that's perfectly normal for an approval test. The message it's giving us though is that there's a problem. This machine has no reporter configuration. We would like to do a little bit more setup so that it would present us with a better error message. I'm going to edit my configuration and add some additional arguments for pytest. I need to tell it which reporter to use to report the failures. I'm going to use the simplest one, PythonNative. So now when I run it again with this configuration, it is presenting me with a diff. And as I said, this is the simplest kind of reporter, and it's just showing a kind of textual, ASCII art kind of a diff. The files are on my disk though, and it's also telling me, look, this test has failed because these two files don't match. It's just created those two files when it ran the test, and it's this received file that I want to look more closely at. On the right, what my printer has produced, the receipt with the apples and the the total. And on the left there is all these assertions I had before. All those things I was checking in the assertions that the total price is within 0.1 of that number, that there are no discounts, that we have one item, it's apples, it's got this unit price, this quantity, and this subtotal, all of that is being printed out in the printout of my receipt. I'm actually going to open up a diff view showing the received text on one side and the approved text on the other. Now, approved is empty because I haven't approved anything yet, but that's what I'm about to do. I think this text looks good, so I'm going to approve it. There. Just using the diff merge tool to transfer the text into the approved file so that now they're identical. If I run this test again, this time it passes, and that's because the text that it received when it ran the printer on the receipt exactly matched the text that was in the approved file. Now, I don't need all of those assertions anymore. I'm going to rely on this approval test. If any of those assertions would have failed, that would actually cause my printout to be different. And that would mean that the test would fail because the text wouldn't match. Now, it's actually perhaps not so convenient to have the approved file living in the same folder with the test code. I would like to have them in a subfolder. So I'm just going to create that folder for approved files, and I'm going to configure approval tests with a file, approvaltest\_config.json file. And in this, I'm just going to tell it where it should look for approved files in the future. I want to use this subdirectory. So now when I run the test again, it actually fails because it can't find the approved file. And it's assumed this is a new test and created me a new approved file and a new received file. So I can just move the approved file that I had into the right place, and then the test passes again. But I don't think I'm quite done yet. There's a problem with this test. It's called 10% discount, but I don't actually get the discount. I think I would like to have two tests, one where I don't get the discounts and one where I do. So I'm going to copy this one and rename the first one to no\_discount. Then in the second test here, I can actually buy the toothbrushes that have the discount. Okay, so now both my tests are failing again because the received text doesn't match the approved text in either of them. So looking first at the 10% discount test, it's pretty hard to read that diff on the command line. So let's have a look at it in actually a diff merge tool. Okay, so here you can see on the right is my received text. There's my toothbrush, and there's my 10% discount. And looking at those numbers, that looks reasonable. On the left, of course, is the text I previously approved with the apples. And because I changed it now to buy toothbrushes instead, that text no longer happens. So I'm going to approve this new result using the diff merge tool, and that one should pass now. But let's have a look at the other test. And again, this test didn't have an approved result. It uses the test name to find the approved file. And because this test name didn't exist before, there was no approved file and it's empty. So again, we need to just approve this result by transferring the text into the right file. So now when I run these tests, they're both passing.

Using Approval Testing Appropriately

We've been through this before. Tests usually have three parts, arrange, act, and assert. In this approval test, we replaced the assert part with a diff. Check if what happened matches the previously approved version. And if it doesn't, the test fails. At that point, we want to be able to easily work out if the difference is because of a bug that we should go and fix or if it's a good change that we want to approve. So we can't just use any old text for this. That's why we invest in developing a printer. The printer displays the test results, often domain objects, as text. The layout is designed to give a readable diff when the test fails, so you can see the context of what's changed and make well‑informed decisions about what to do. You can often reuse the same printer in more than one test case. So when you're designing a test case, you will still need arrange and act, but you can choose whether to do a normal assertion or a print and diff. There are advantages to the print and diff method. I think it's easier to understand test failures. Instead of reading an assertion failure message, I can look at a diff that shows the change in context. If you actually prefer the new result, it's easy to update it with a diff merge tool. Now there is, of course, a cost associated with developing a printer, but it shouldn't be significantly more than the cost of developing assertions. And I think this approach is often cheaper overall. This is a picture of the test pyramid and a link to an article by Martin Fowler that explains more. The advice is to have lots of small unit tests because they're relatively cheap and fast and fewer service and UI tests which test bigger pieces of your application because they're slower and more expensive. I generally agree with this advice. And if you're writing small, fast unit tests which each assert one thing, you probably don't need approvals. Some of your tests though will be higher up in the pyramid, testing bigger chunks of code. These pieces will have more behavior, and you will probably want to check more than one thing. In that case, it could be a good idea to use approval testing instead of multiple assertions. Some more advice. Printer design is actually the most important topic for successful approval testing. You need to lay out your results on multiple lines and exclude irrelevant details so your tests only fail when they have good reason. If the test fails and you don't understand from the diff what's going on and whether you should approve it or not, that's a sign you need to improve your printer. Final piece of advice. Use tools. Don't create or update approved files by hand. Use an approval testing tool like this one. Don't build your own. There is a lot of experience and wisdom built into this tool by the community that maintains it.

Demo: Code Coverage

The next topic is code coverage. Let's look at a demo. Here, I'm back in the supermarket receipt project, and these are the two tests that we wrote before. Both these tests are still passing, but I've done a little refactoring since I last showed you them. I've extracted quite a lot of test fixtures into this conftest file so that my tests don't contain as much duplication. I'd like to run these tests with coverage enabled so I can see which lines of code are being executed. I'm going to add coverage as one of my project dependencies. I'm going to show you how to use it on the command line first. So I need a command line that will run all of my tests, and here's one I prepared earlier. It's going to run pytest with the right flags and the right Python path. Now I should be able to modify this command line so it will run them with coverage. I just need to exchange the Python for coverage run. And that will execute the same thing, but it will gather coverage data. So now the coverage tool has written a little file of data about the coverage, and I want to see the report about what it's discovered. On the command line, I can just write coverage report ‑m, and it shows me here a list of all the files in my project and which lines of code were executed effectively. It's actually showing me the lines that weren't executed in this list of missing lines and the overall percentage for each file. Looking at this, I can see I may have a problem in shopping carts since only 71% of the lines are being executed when I run my tests, there's a lot of code there that I might like to have some test coverage for. It might be convenient to see which lines more graphically. So coverage also comes with an HTML report. So this is the same information I saw on the command line, but now I can click into each of these files to see specifically which lines are being covered. So here in shopping carts, the lines of code highlighted in red are not being executed when I run my tests. And in this handle\_offers method, I can see there's quite a few lines that are missing coverage and therefore all the various different kinds of special offer. I actually only have tests for the 10% discount. In some IDEs, you can get the same information integrated into your editor. This is PyCharm. And when I run with coverage here, it presents the same kind of reports with the same numbers as before. And when I click on the file, it actually opens it in my editor, which is kind of convenient so that I can edit the code and see the coverage information in the same place. This time, it's not highlighting the whole line. It's just putting this red and green marker in the gutter next to the line numbers. I think I'd better write a new test for three-for-two special offers and see if I can improve my coverage. I'm going to base the new test on this existing test. Of course, I'll need to rename it after the scenario that this is. And this time I'm going to do the three-for-two offer, and I should probably buy three toothbrushes so that I actually get the discount. Of course, this test fails the first time I run it because it's an approval test, and I need to look at the output and see whether I can approve it. And looking at this, I can see the three toothbrushes that I bought, and then there's the three-for-two discount, and the total appears to be correct. So I think I can approve this. Now I should have a passing test. I'm going to run it again with coverage. And here in the headline report, I can see the coverage has gone up to 77% now, and those lines of code that are to do with the three-for-two offer are being covered. They're being executed when I run the test. As we've seen, a coverage report shows which lines of code are executed by your tests. If the line is not being executed, it could have a bug in it, and your tests wouldn't be very likely to find that bug. It's often useful to look at which lines are not being covered by your tests. It can help you to see where you don't have enough testing, and then you can make informed decisions about whether to invest the time to write those tests.

Demo: Combination Approvals

Combination approvals is a technique that can help you to get good test coverage with only a small amount of test code. Let's look at an example. This is the gilded\_rose refactoring carter. At its heart, it's a piece of rather complex logic for updating item properties. An item has a name, sell\_in, and quality. The update\_quality method uses the item name to help decide how to update the sell\_in and quality values. I've already written a test using approvals for this method. I've written an item printer that shows the name, sell\_in, and quality. The test passes, and this is what the approved file looks like. This is all very well, but the coverage unfortunately is not that great. As you can see, I'm not actually executing large parts of this logic. I would like to improve this test, and I'm going to do that using combination approvals. I'll begin by just refactoring the existing test a little bit. I'm pulling out as variables the things I'm going to need to vary to access all of the logic in this method, the name, the sell\_in, and the quality. Then I can extract this part of the test as a function. I'll name that update\_quality\_for\_item. And those three things that I extracted as variables are now parameters to this method, and I want them in the same order as in the print item. This test still passes. That was just a refactoring. Now I'm going to introduce verify\_all\_combinations. This is similar to verify, but it takes more arguments. The first argument is the function that the test should call, update\_quality\_for\_item. The next argument is a list of all of the values it should pass as arguments to that method. So that's different values for the name, the sell\_in, and the quality. So I'm just going to make those three things into lists that I can pass here. There's also a third argument, which is the printer basically for each of the test cases. So this is very similar to the printer I already have, except it takes two arguments. The first thing is the the specific values for this scenario for the name, sell\_in, and quality. That's the args. And the result, that is the result for when you've called update\_quality\_for\_item, and that will be an item. So what I want to do here is to print out the arguments that I got and then what happened after the method, what the printed item now looks like. So this test case does basically the same as the previous one, but it doesn't actually pass because I actually have a different printer, of course. So now my printer is also printing the arguments that I pass in. That's being printed there. So that's why there's a difference. But it's the same test case essentially. And the coverage is still the same. I would like to improve this coverage. In order to reach this branch of the code, I'm going to need a quality that's greater than 0. So let's add that to my list of values for quality. Now the test fails. I have additional output for the case when the quality starts at 1. The formatting needs improving though. I'm going to put a new line on the end of my printer so that now I get separate lines for each scenario. So I have one test case effectively where the quality is 0 to start with and it stays 0 and another where the quality starts at 1 and then goes to 0. So, I'm just going to approve that. Whatever this code does can be approved to actually be correct. And now I can see that I've got the coverage that I wanted of that line of code where the quality is greater than 0. Now I'd like to cover this else clause. And for that, I'm going to need various different names of item. So I'm going to add some more values of these special names for particular kinds of item that get treated differently in this logic. So I've added three different item names there, and the test fails because I changed it. And you can see here in the printout that I didn't only get three new lines in my output. I actually got six. It's testing each of these items with all the combinations of the other arguments that it knows about. So I get six new values. Again, I'm just going to approve this. Whatever this code does can be assumed to be correct. And now I'm just going to check the coverage again. Did I get that else clause that I wanted to get? Yeah, looking good. So there's still a few lines here that I'm not covering. And I think to get them, I'm going to need an even higher quality. So I'm going to go back here and add 2 to my list of qualities. So of course, the test fails because I have new outputs. I've got four new test cases for the door, four different item names, and I'm going to approve that and check the coverage. And yeah, I seem to have full coverage now. And if you notice, I needed an extremely small amount of code to get there, and I have actually 12 test cases now.

Using Combination Approvals Appropriately

I just showed you combination approvals. You identify some data that needs to vary in order to cover all the parts of your logic, and you test all the combinations of it with an approval test by printing the output. I'd like to illustrate this visually. If I have three parameters that need to vary and two available values for each, then there would be eight different combinations. If I add 1 additional value for one of the parameters, the number of combinations goes up to 12. So the number of combinations increases quickly with the number of possible values for a parameter. In this example, the number of available variants for the third parameter increases to , and the number of combinations went up to 24. Now this can help you to get good test coverage as we saw with very little additional code, but at quite a cost in the number of test cases and the time to run them.

Demo: Mutation Testing and Branch Coverage

Back to the demo. I want to look more closely at how good my combination approval tests actually are by doing some manual mutation testing and checking the branch coverage. With these combination approval tests, we have 100% statement coverage of the update quality method. Now that sounds impressive, but how good are these tests at actually finding bugs? I'm going to do some mutation testing, deliberately introduce some bugs and see whether my tests find them. First up, a bug on line 19 here. Yep, my test failed. That bug was found. That mutant was identified. This one too. But what about this code on line 28? Ah, now that is what I was looking for. These tests have not caught a bug on line 28. I'm going to mark that as a mutation that survived. What about this line? Yep, same thing. That one is also not being found by my tests. Okay, what about here? No, my test found that one. That was good. What about this one? Oh, now that was also not found, that bug. And one last bug here. Yeah, my test found that one. Okay, so I've put in maybe half a dozen bugs, and I've found three bugs that were not found by my test suite. The next thing I'm going to do is improve the kind of coverage that I'm running. I'm going to enable branch coverage in this tool. If you're using the command line, there is a flag for that, ‑‑branch, it takes longer to run the coverage tests now. It's more computationally intensive, but I get more information too. Some of the lines that were previously green are now yellow, and that means they're only partially covered. This helps me to understand how some bugs were not found by my test suite. They're close to lines that are not fully covered. I'm going to do some more work now with adding more combinations to see if I can get 100% branch coverage of this code. I have a feeling I need a quality that's much higher to catch that line 43. This line 37 here or line 38, I think what I need here is a lower sell\_in. Yep, that seemed to catch that one. Catching these lines, I think I'm going to need some sell\_ins of those boundary values of 6 and 11, and possibly also I need another quality that's quite high, 49. Okay, so now I think I've achieved 100% branch coverage of this code. The test is currently failing because it didn't approve it yet, and I've gained a lot more combinations. I'm up to 80 cases. I'm going to approve that and see whether my tests are any better at finding bugs now. I'll reintroduce this bug. Yes, that's what we wanted to see. The tests have found that bug now. What about this one? No, actually, my tests still haven't found a bug on this line. What about the last one? No, we're not finding that one either. So even though I actually increased my test quality and coverage substantially, there are still bugs that I can put into this code that my tests don't find.

Using Coverage and Mutation Testing Appropriately

I just showed you mutation testing, which is a way of assessing how good your tests actually are. Beginning with passing tests, you deliberately insert a bug. You make a mutant of your code. You run the tests and see if they fail. And if they do fail, that's a good sign your tests have killed the mutant. But if they don't fail, then that could be a problem. The mutant has survived, and it means your tests could be better. It's worth bearing in mind though that tests and testing are never going to be perfect. I showed you how to do mutation testing by hand, and that can be a useful thing to do, but there are tools that can do it for you. However, they only know about certain kinds of bugs to insert, and all mutation testing is time and resource‑intensive, and it's not something you can afford to do all of the time. Another thing I showed you in the demo was how to enable branch coverage. If you have a conditional in your code or like an if statement, then the flow of execution branches, and it chooses one or the other depending on whether the condition is true or not. If the condition is always true in your tests, when you run coverage, then the true branch is marked green, covered, and the other branch will be marked red, not covered. If you enable branch coverage measurements, then the actual line of the conditioner will be marked yellow, covered but with only one outcome. This is particularly helpful when there is no else clause, which is what happened in the demo. The branch that is always taken is marked green. But since the else clause is missing, no lines are marked red so you don't see the problem until you enable branch coverage, and then you get this additional information. The line containing the conditional is only partially covered. Test coverage and mutation testing are useful tools so developers can make informed decisions. You can use it as we did to spot missing test cases and decide where to invest in more tests. However, even 100% branch coverage and excellent mutation testing scores does not mean your code works or has no bugs. It can be a useful source of information though, especially if you track trends over time to help you monitor whether you're continuing to add the tests you need. It's unrealistic for most projects to get 100% coverage. Some teams have a coverage target, perhaps 80% for the project as a whole. And if the coverage drops below the target, people start asking questions and prioritizing testing activities, which sounds like a good idea. But in practice, having a target like this can go very wrong. What I've seen being problematic is when somebody outside the team, like a manager, punishes the team if they don't meet the coverage target. And it's actually quite easy for developers to just game the system and increase the coverage numbers with really bad tests. Mutation testing can help to show if that's happening, but the numbers you get from those tools are actually really hard to interpret and use well and might just make the situation less clear.

Demo: Pairwise Testing

One final technique I'd like to show you, pairwise testing. At the end of the last demo, we still had two mutations that weren't being caught by the tests. So I've done a little bit more work now and added some more combinations, and now I am catching these mutants as you can see, but at a cost. I have far more combinations now. I'm actually up to 120 different combinations, but I do have pretty good coverage now. The thing is for a real project, that's probably going to be too computationally intensive to run all those tests all the time. I wanted to show you all pairs. Instead of calling verify\_all\_combinations, I call verify\_best\_covering\_pairs, and that makes a selection out of all of those combinations. It's cut down that 120 scenarios to just 30, which are optimized for the right kind of spread of different combinations of values. So I'm going to approve this new result to get my test back to passing. Then I'm going to have another look and see whether it still kills those mutants. So yes, that one is still being killed. But actually, that one is not. So by reducing the number of combinations with the best covering pairs, I still have very good test coverage. But here running with the branch coverage enabled, I can see that that line of code is only partially covered still. But still, it's very good coverage and at a much lower cost of only 30 combinations rather than 120. Going back to this picture which we saw before, if all combinations gives us 24 test cases, all pairs will select just 12 of them to give the best coverage of pairs of different available values. It gives you nearly as good tests, but with less computation. So to summarize, combination testing improves your coverage, approval testing makes these tests easier to manage and update, and all pairs will reduce the number of combinations you actually need to run.

Module Summary

In this module, we looked at approval testing, which is particularly helpful when your unit is a little larger and you want to assert more than one thing. We looked at measuring code coverage, which is useful for finding out things that are not tested. And then with all combinations and pairwise testing, we can increase the code coverage. Mutation testing can also help you to assess how good your tests are.