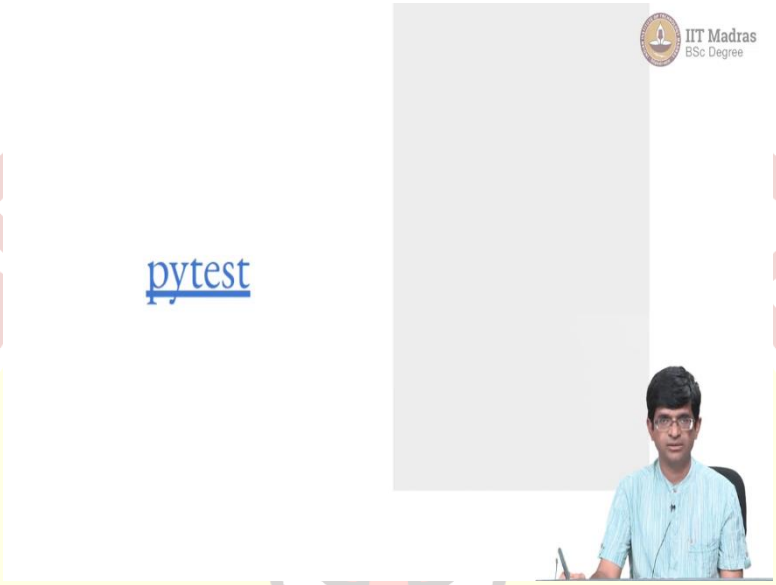
IIT Madras

ONLINE DEGREE

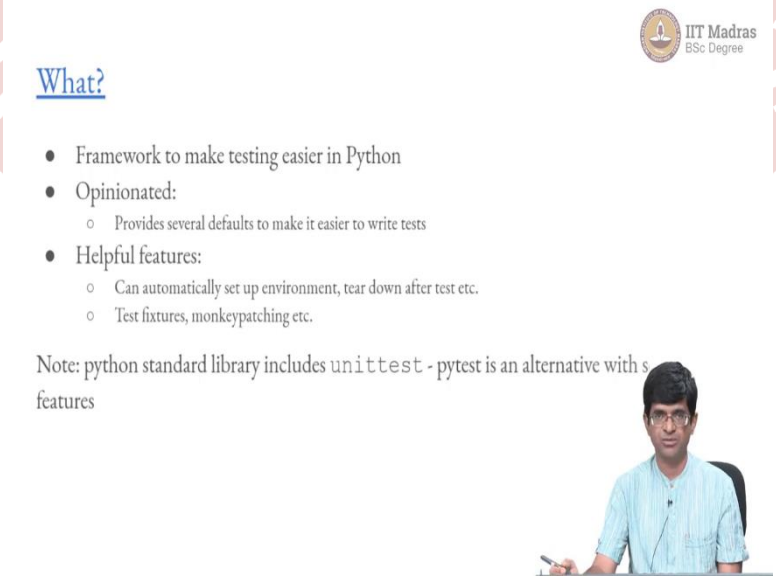
Modern Application Development – I
Professor. Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Pytest

(Refer Slide Time: 00:17)



Hello, everyone, and welcome to this course on Modern Application Development. So, with all this background on, what is testing? How do we generate tests and so on place, let us look at a specific example of one framework that can be used to implement tests in the Python programming language. And what we are going to use is a framework called pytest,

(Refer Slide Time: 00:36)



Now, it is a framework that makes testing easier in Python. It is somewhat opinionated in the sense that it provides several defaults, meaning that, if you just write a test in a certain way, automatically, it will pick up that this is a test and it will run it, it makes it a lot easier to write tests. On the other hand, there may be cases where people disagree with it, where they feel that this is not really the way that I should be writing the code.

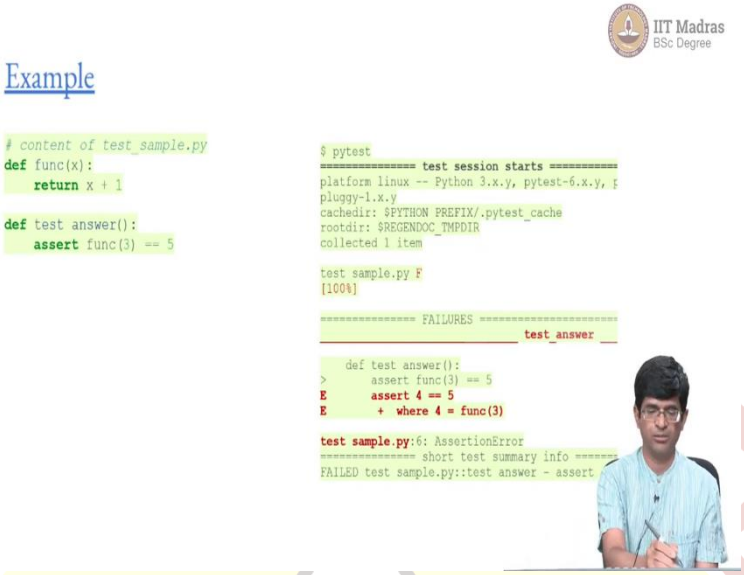
In some cases, you can change that in others, it just sort of says, look, you write it this way, if you want, pytest to be able to handle the tests. And very often, what you will find is that, many of these sort of strongly opinionated frameworks can be helpful in the sense that they reduce the chance of two people just assuming different things about the software.

Now, pytest has a number of helpful features. It can automatically set up an environment, tear it down after the test and so on. In particular, it can put up something called test fixtures that we will look at later. There is also something, which is sometimes called monkeypatching, which essentially is trying to sort of make modifications to the code in order to run a particular test that we will not really be getting into, but it supports those kinds of activities as well. There are ways by which it helps you to do that. At least the test fixtures and various other things, we will look at a couple of examples moving forward.

Now one thing to note over here is that the Python standard library actually includes something called unit test, which is a Python standard library element. Pytest is an alternative. In a lot of cases, you might be able to get by just by using unit test itself. Pytest basically has some additional features and because of that, we will be focusing on it over here.

But what I want to make clear is that pytest is not the only way of doing it. It just so happens that it is something just like flask, it is an option that we have considered to be a good option and that we are using as the basis for discussion here. You might have something else that you prefer, in which case that is perfectly fine. It is just that for this course, we are going to be using pytest.

(Refer Slide Time: 02:48)



So, let us look at an example. Now assume that I have a file, which I will call test_sample. py, and this is the content of that file. This is just an example from the pytest website itself. So, what it does is it defines a function, func(x), which just returns x+1. So, if I call it a func (2), I should get the answer 3, if I call func(3), I should get the answer 4, and so on.

Now I also define one more function called def test_answer, the underscore is not clearly visible here, but it should be there. And what that does is, it basically does an assert. And assert is a Python statement that basically takes a conditional or some expression. And if the expression is true, it says, Okay, if the expression at false, it will basically raise an assertion error, it will fail at this point.

Now, in this case, the test that we are writing is actually a bit bogus, because I am asserting that func(3) should be equal to 5, which I know is wrong. Func(3), based on what I have written should actually be equal to 4. But so there are two possibilities. One is either my test was wrong, or my code was wrong. Because if I assume that the test is correct, and that what I wanted was that func(3) should be equal to 5, then maybe my code should have been written x+2. But there is also the possibility that I just wrote the test wrong.

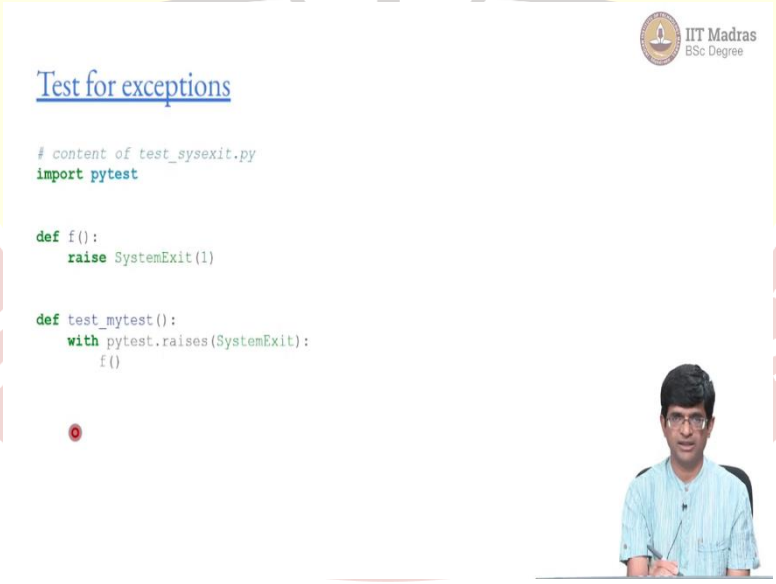
So, of course, you need to decide which one is correct, which one is the actual right thing over here. Either way, what this does when I run pytest, with this test_sample.py if I just run the command pytest, what it will do is, by default, look at all the Python files that are there that start with the name test_, so that is what I meant by the opinionated part. It sort of assumes that anything which starts with test is a test file.

Now within that, once again, there are two functions, but only one of them this test _ answer will be considered a test. This func that I have declared over here will just be considered some additional, something that is required for the test_answer to be executed. So, what it does is it starts the test session, it gives you some general information, collected one item, what is this one item, test_sample.py within that it has one test.

And what it says is, it comes up with this f over here, which says that one test failed. And it also says 100 percent which means that it ran 100 percent of the tests that were present over there. In this case, the result was that it failed. The good part is, it also gives you detailed information about the failure.

So, what exactly happened in this failure, it says that, the assertion required that func(3) should be equal to 5, but what happened in practice when I ran the code is that I got 4 on the left-hand side, which is not equal to 5 on the right-hand side, where 4 is equal to func(3). So, basically where did this 4 come from it is equal to func(3), it is not equal to 5, therefore, there was an error, and it failed. So, this, as you can see is a very trivial example. Now I need to go back as a developer and decide, was my test wrong or was my function wrong?

(Refer Slide Time: 06:29)



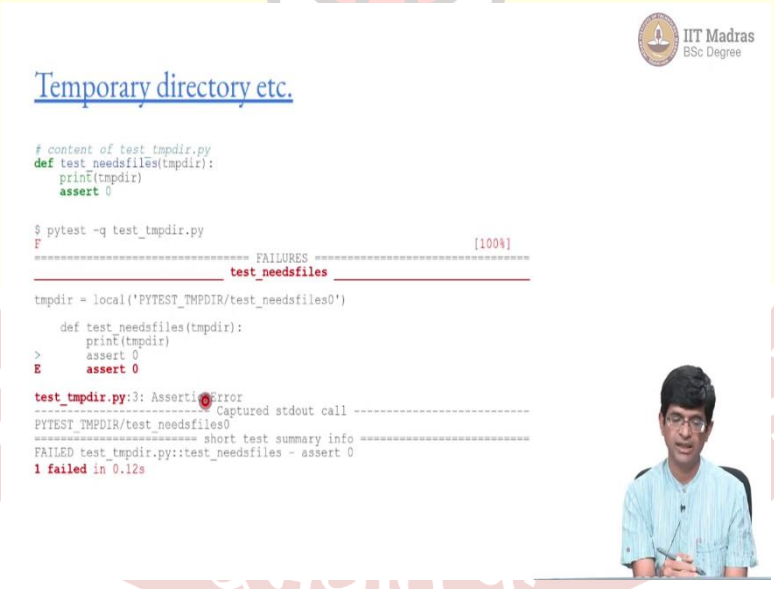
You can do more with this. For example, one of the things that you could do is, supposing a particular function when I call it is supposed to raise a particular exception, in this case, a SystemExit exception. My test over here because it starts with this word test the test will say, with pytest dot raises SystemExit f, which basically means, that it is going to check whether when I run the function f, does it raise SystemExit? In this case, it does, so it is fine.

Now this is obviously a contrived example. But supposing I wanted to have this thing and check a test where let us say I am trying to access a particular data structure or an array outside of its bounds. So, it has, let us say, it is maybe an array with 10 elements or something else of there or a particular index number of indexes, and I try to access the hundredth element in it.

It would use some kind of, it would raise some kind of an error, some index error or something like that. And I might write a test that says, If I access it with something, which is outside of the bounds, it should raise index error. So that is one neat way by which I can just basically have a test which makes sure that when I run the function with certain inputs, it will raise the right kind of error.

So, I could say, with pytest.raises(SystemExit) f(3), so maybe 3 should have raised an index error. Whereas, I might have another test that says, check for some other kind of thing, that is, as long as access correctly, it should not raise an index error. So, all those kinds of things can be coded into tests in this way.

(Refer Slide Time: 08:23)

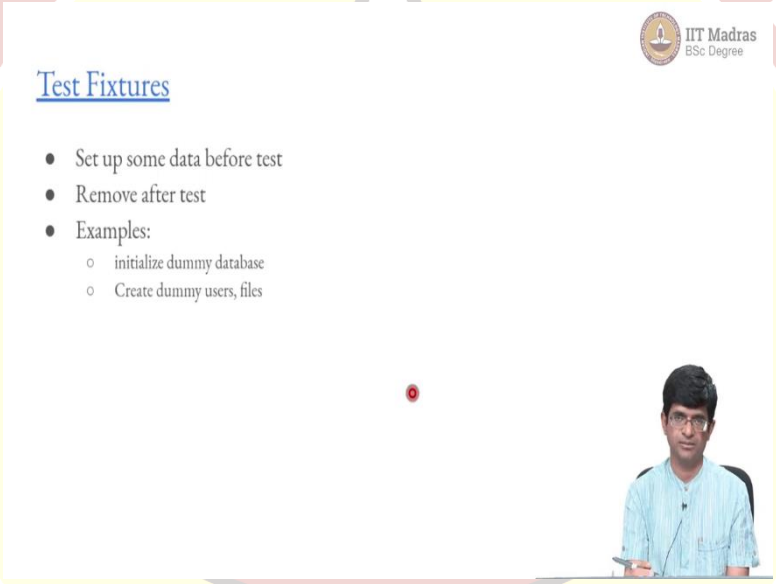


There are many tests that require you to sort of create temporary files, create temporary directories and so on. I could define a test where I basically tell it that I need a new temporary directory and, in this case, I would actually need to specify how the tempdir has to be done that is it needs to be created. But once I do that, it would actually notice what I have done here I have just done assert 0.

Assert 0 is one way of guaranteeing that the test fails because 0 is basically the same as false. So, when you do this automatically, it just fails, but it also prints out everything else that it had over here. So, it prints the tmpdir, for example. And it says that it came here and it finally failed because of the assert 0.

So, the captured stdout is this value pytest tempdir lash test needsfile0, it basically created the temp folder for you, which can be done automatically. The bottom line is pytest is taking care of creating an extra folder for you guaranteeing that that folder does not sort of they accidentally delete or remove anything which already exists, so that you can use it for some temporary thing and after the test is completed will automatically remove that folder.

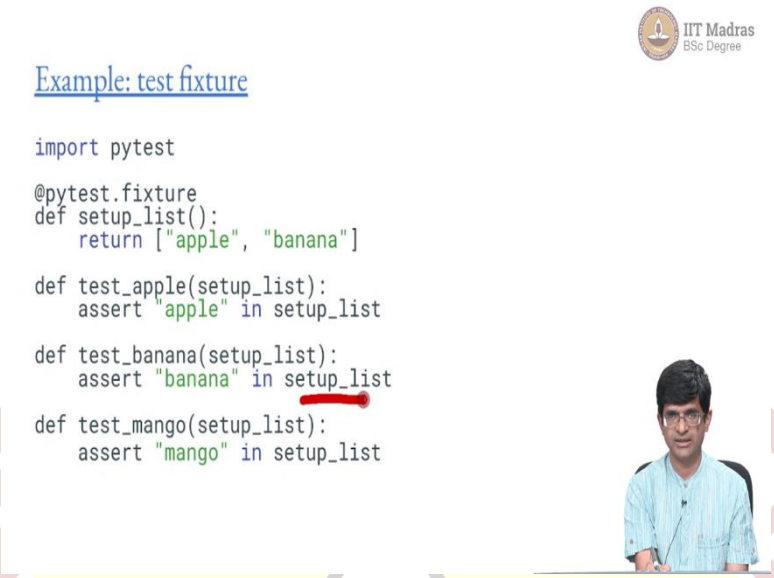
(Refer Slide Time: 09:49)

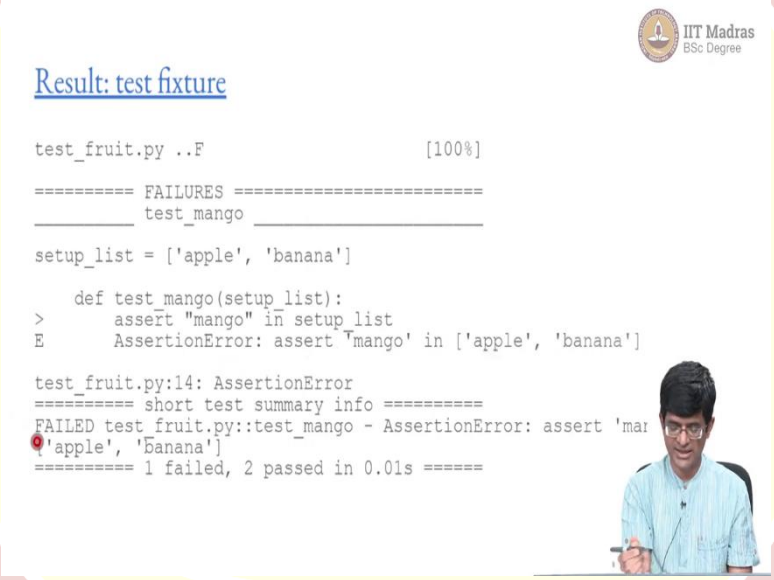


All of which sort of relates to this concept of, so called, test fixtures. A test fixture is something where I need to set up some data before running a test and remove it after running the test. So, you might recall that, earlier we were talking about what happens if I want to add a student to a course and give them marks. I need to be able to create a dummy database that has the right information.

It has one student, it has one course, so that with that database, I can then access the controller and tell, add the student to this course and check whether it did the right thing. So, on the other hand, I do not want to write all of that code inside my test function. Especially because I might have multiple different test functions that need to work with the same database. What I can do is define something called a test fixture. And the test fixture is something that will get automatically created or run just before the test itself is run.

(Refer Slide Time: 10:53)





So, this is an example of a test fixture. Once again, a sort of contrived example, but what I am doing is, I import pytest and then I have one function out here, this def setup list, which basically just returns a list two items in the list. But I have declared this function, I have decorated this function as a pytest.fixture. Which means, that if I try calling setup list directly in the Python code, it will give me an error saying that, look, you cannot call this function directly. It is now a decorated function, which means that it is meant to be called only as a pytest fixture.

Now I define three tests, these three functions out here are all different tests. The first one says test apple. And one parameter that I pass into it is set up list, which is basically the name

of my test fixture. So, what happens is, when I run the test apple test, before going inside this, it will first run this setup list function. What happens?

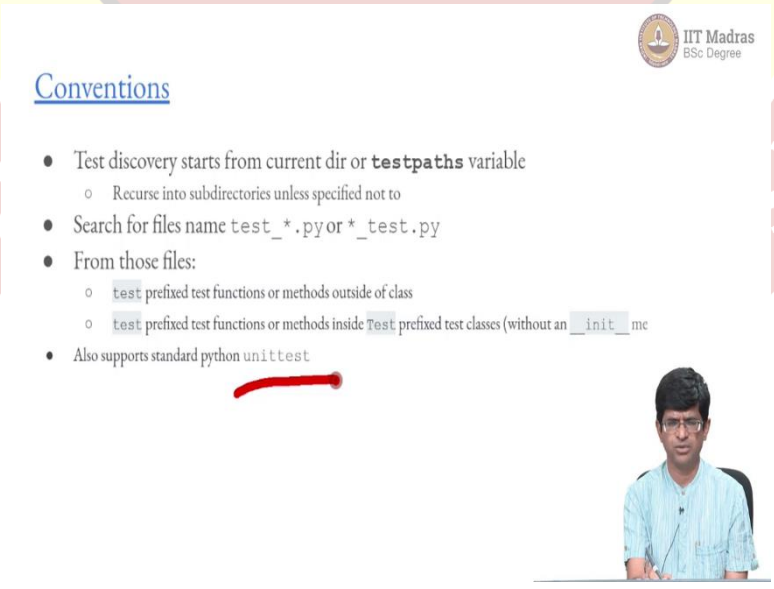
That will return a list which contains the words apple and banana, and will be set to this variable setup_list. What is a test do? It basically asserts that the word apple is present in the setup_list? Similarly, I have another test, which asserts that banana is present in the setup_list. And our third test that asserts that mango is present in the setup_list.

What happens when I run this entire thing using pytest? What I will find is that it basically gave me back this output, ..F. So, what does this ..F as the result of test fruit dot py indicate? It means, that the first two tests passed when the third one failed, and it gives me more details. Test mango failed.

Setup list in this case is equal to apple banana, and assert mango in setup list failed assertion error. So, it failed this test. In other words, it is telling you that everything happened the way that I expected. I had set this up as a test fixture meaning that before I get into the test, I need to update this list with these values.

And once I get into the test, it will just use that. And then once I am done with it, the variable is gone again. So, in fact, if I modify the variable in any way or here, that will not affect the other tests that I create.

(Refer Slide Time: 13:32)



So, test fixtures can be used for a lot more, which we will get to in a moment. There are a few conventions that pytest follows. For example, the test discovery basically starts from the current directory or there is another variable called testpaths that can be used. Usually, it will

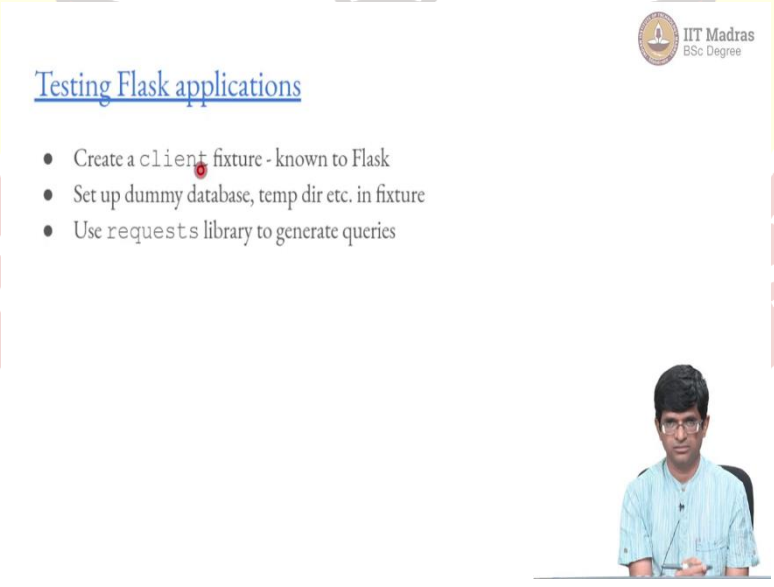
recurse into the sub directories, meaning, that it will go into the sub directories and look for any file name, test something .py, and try and run tests within that.

What it is basically going to look for is files that either are named test underscores something .py or something_test.py both of those it looks for. And inside those files, if there is any function or a method, which is starts with the word test, or if there is a method inside a class and the class itself starts with a test with a capital T and the function starts with test, then it will be all treated as a test.

Now, you do not need to learn all of this by heart, you need to sort of, get the big picture of what is happening over here. The main point is that pytest has some standard conventions it follows in order to look for tests. What are those conventions? At the end of the day, rather than trying to remember all of these different things, it is probably best to pick a set that works for you and follow the same principles throughout.

And finally, the good thing about pytest is it sort of meant as a superset of the unit test standard library functionality, which means, that if you had written tests earlier using Python unit test, they will also work with pytest, But pytest gives you additional things like the test fixtures and various other things that probably make it a bit more useful.

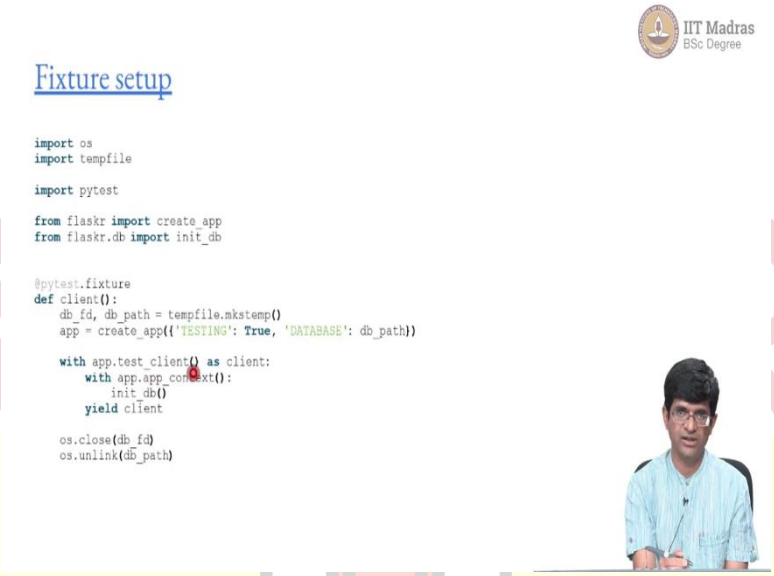
(Refer Slide Time: 12:45)



Finally, with all of this in place, how do we test a flask application or a web application? Now, flask provides good support for using pytest. In particular, it allows us to create a fixture, which is called a client. And this client can be used in order to generate requests to the flask application. It can do other things. So, the in the fixture, I can, for example, set up

things like dummy database, temporary directory in which to run and so on. But ultimately, it sort of uses the request library from Python with this client object in order to generate the requests.

(Refer Slide Time: 15:59)



Let us take an example. The first thing is how do we set up the fixture? So, all of this part of it, these are the standard imports. The assumption we are making is that our application, instead of application.py, it is actually in a module called flaskr. So, the creat app for example, would be inside the flaskr module. And similarly, I also have a db within that. So, all of this is fairly standard flask terminology.

You may not have used exactly the same kind of setup in the examples that we did earlier in the course, but this is just a slightly different way of implementing exactly the same thing. Bottom line is that flaskr is the name of the application that we are creating and it has an init_ db and a create_app, which are very similar to the things that we used already in the course. The important thing is, client, which is defined as a pytest.fixture over here.

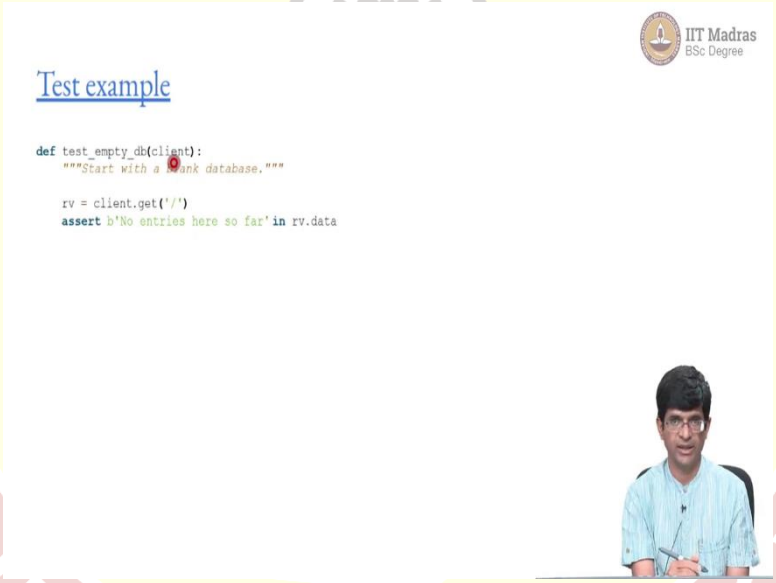
And what it does is, it basically creates this temporary file. And what we are going to do is, we create the app, along with setting an environment variable testing to true, and we give the database as this temporary path that we just created. Now, this will work assuming that the database is some kind of an SQLite database. And that when you do init_db(), it is able to take this path and create a database or that at that path.

If it was MySQL, for example, you may not really be able to do the same thing. Now, the interesting thing that you will see or hear is this command called yield. In case you are not

familiar enough with Python to know what the yield means, for the time being, just ignore it. For our purposes, we just what it means is that it has created something that can be used in a different context. So, effectively, it is saying that the code has run up to here and then is sort of letting this client variable go back to the original code, where you can do lots of things with it.

But at the end, when you are done with all your tests, and so on, and you are sort of willing to close the client, you will come back over here, close the database, file descriptor, close the path, and basically do the cleanup so that the temporary file and path that you created are deleted. So, this client, in other words, becomes something, it is something that is part of flask itself. And at the end of the day, it is creating this app.test_client, which is a function within flask, and it is allowing you to use that for your test cases.

(Refer Slide Time: 18:49)

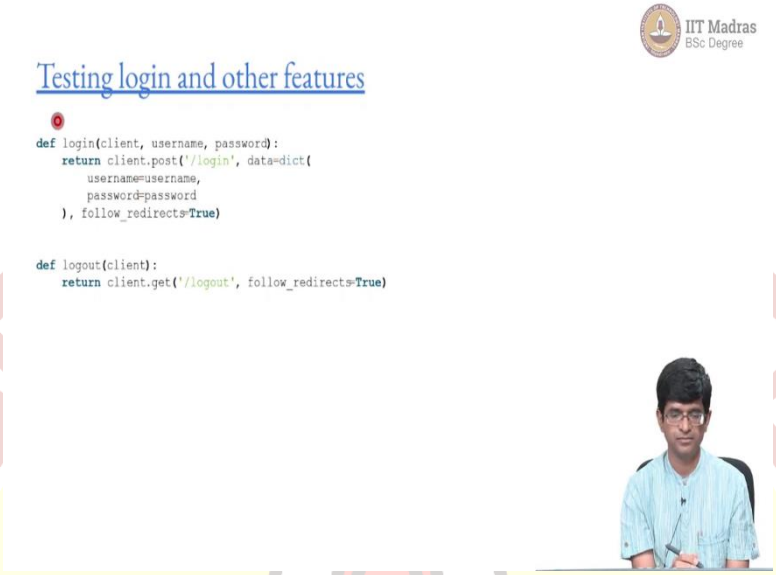


How would you use it? This is an example. I might, for example, start with the test empty database. And what it does is, it just now this client also has certain methods that can be called one of them is client.get, which will perform a get request. Client dot get to the / endpoint in this case, I want to check and verify that I get back the text, no entries here so far in the response.

So, rv is the variable that takes in the output from client.get, rv.data contains the actual text that goes returned, rv.header will probably contain the status codes and so on. And in that I want to make sure that there is this particular text. So, this is an example where I am just basically creating a blank database testing that nothing is present in it and so on. But you can

imagine that, now that I'm able to create a database and the client and get requests, I can now start testing any endpoint that I wanted.

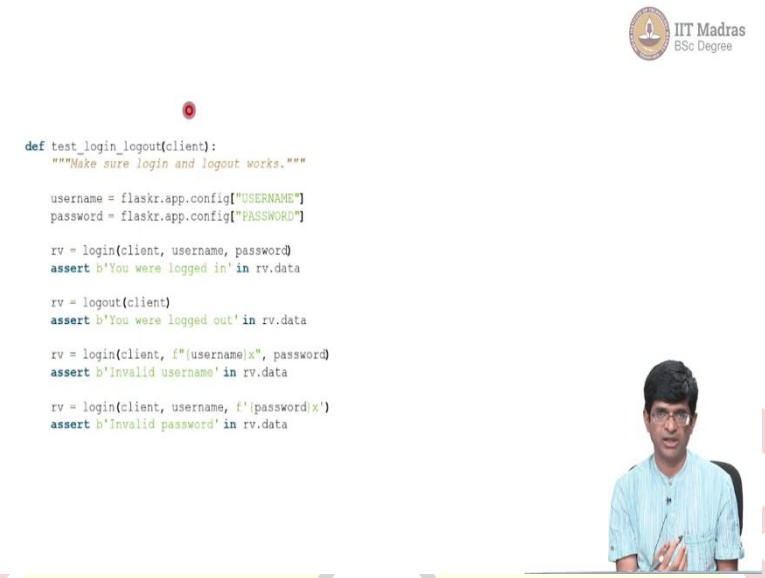
(Refer Slide Time: 19:55)



I could go further, so I could define other functions. For example, login, logout and so on where I can basically say a login function would do a client.post with this username and password that I specify, but has one extra feature which basically says follow redirects. Meaning that, it would not only do the client.post, it will also see what response it gets. And if the responses are redirected to another page, it would not follow that as well and finally get me to the point where I have logged in or not logged in.

Now, this login function is not the login controller that you would normally use. This is clearly only for testing because it uses the client test fixture and is actually calling the / login endpoint that I have. So, this login and logout are functions that are present inside my test fixture file, inside my test file. But because they do not start with the word test, they are themselves not considered test cases, they are just helper functions. How would I use something like this?

(Refer Slide Time: 21:05)



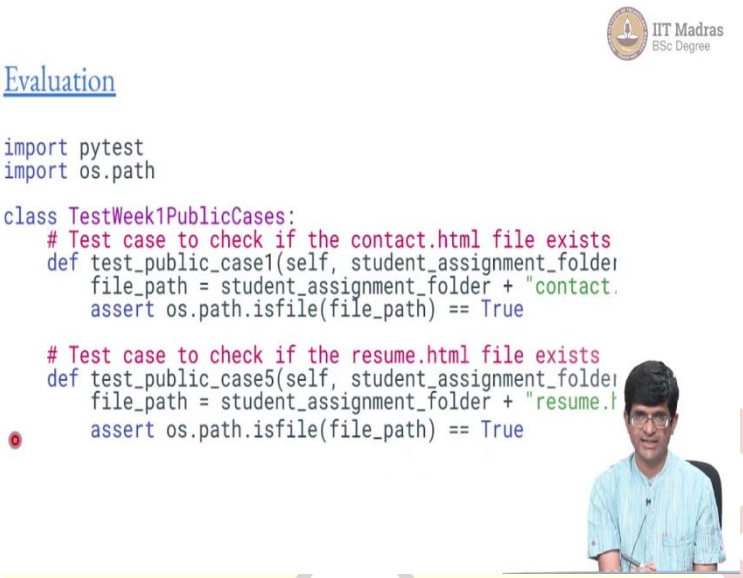
I could now actually define a test case, where I say, look, pick up the actual the username and password that has been configured. Try one test, which is login using the correct client username and password and say, okay, the rv.data should contain the words you were logged in.

Similarly, test logout and it should contain the words you were logged out. But now, if I try logging in with an invalid name. See, I basically appended an x over here to the username, I should get the response invalid username. And similarly, if I try with an invalid password, I should get the response invalid password.

So, effectively, what I have done is, I have created a test over here, which will test for different conditions and it uses this login, logout temporary functions, helper functions that I have created, all of which make use of this client text a test fixture.

So, broadly, what I am trying to get at over here is that you can use this kind of structure in order to set up the framework for testing, create some extra functions that will help you with the testing, and actually test detailed functionality of the flask application exactly the way the end users should be able to try it out.

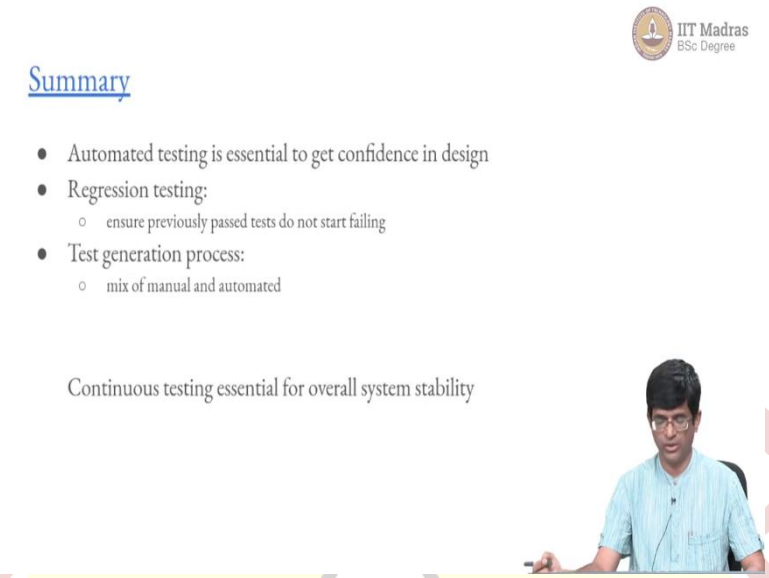
(Refer Slide Time: 22:28)



Now, by now, you would have already come across various parts of the week 1, and various other weeks examples. And if you look at it, we have actually used pytest for even evaluating or automatically evaluating the submissions. And the way that it works is something like this. You basically create the public test cases over here. And within that public case, a test case 1, test case, 5, for example, I mean, there are various test cases that are there. So, the first test is to check if the contact dot html file exists. What does it do? It basically checks that there is actually a file called contact.html inside the student assignment folder.

And similarly, what we have is that there is another one with checks that resume.html is present. So, unless both of these things work, the pytest would automatically sort of fail one or more of these tests. And by running a set of tests like this, we can automatically do part of the grading that is required for the project as well.

(Refer Slide Time: 23:32)



So, this is sort of the overall summary of the whole problem of testing what we want to do with it, and so on. The reason we want to do automated testing is, to get some kind of confidence in the design. We want to be able to capture any regressions, which is that things that were previously working should not now start failing, and we would like this process to be automated to the extent possible, because we do not want somebody to be manually sitting there and testing each line of code.

So, automated testing is essential, so that we can have a certain degree of confidence in the design, but you need to be very careful that you do not sort of overestimated and say that, just because I have 100 percent code coverage or statement coverage, that does not mean that my code is actually bug free.

And a continuous process of testing is essential for managing the overall stability of the system, and for having a reliable system that you can be reasonably confident will work in production.