**w09-01-ModulesAndUnitTesting-2024**

0:01  
OK, so we're now in week 9 and So what you'll see today is we're introducing basically the last concepts for 995 that is modules and unit tests.

0:13  
Now before we go and look at the lecture material, I'm going to point out the obvious, which is that there is a class test tomorrow.

0:21  
The instructions with respect to the class test are on the My Place page.

0:25  
What I haven't done yet and I will do later today is put on there any instructions that are particular to somebody who has an extra time requirement?

0:33  
So the, if you remember the lab rooms on level 13, there's a small room on the left and a bigger room on the right.

0:41  
So we're going to use the small room on the left for the people with extra time.

0:45  
So that will be put here on the My Place page beforehand.

0:50  
So please read the instructions if you haven't done so already.

0:53  
Have a go at the practise quiz.

0:56  
The practise quiz here is as realistic as I can make it.

1:00  
It's got the same format, it's got the same style of questions, it's got the same number of questions and it's got the same time limit.

1:08  
All right, so the name of the game with anything like this is that you answer the things that you know the answer to quickly 1st, and then that gives you more time to think about the things you're not sure about.

1:20  
If you do that, you'll have loads of time.

1:23  
If you sort of work 123 and you get stuck, that's when you could run into difficulty with not having enough time anyway.

1:31  
So if anybody has any questions about the practise assessment or anything like that, please feel free to send an e-mail or put it on the forum.

1:41  
There's a forum section about assessments and it's there for you to ask questions about assessments.

1:48  
If somebody sends me an e-mail, what I'll do is I'll make it anonymous, but I'll post the question on the forum and then I'll post an answer.

1:54  
OK, great.

1:56  
So that's the introduction to the to the quiz and we'll carry on with the lecture.

2:04  
OK, so in today's lecture we're looking at modules and testing and in particular we're looking at unit testing.

2:12  
There are other types of testing which we will discuss in 993 next semester.

2:18  
First of all, Python modules.

2:20  
What are they and why would you want to use them?

2:25  
Now the basic idea of any sort of programming language is that you want to put your code into blocks that you can then have an idea what is that block going to do.

2:37  
You can test it and then you can organise it in some sort of structure.

2:41  
Now, Python is no different.

2:43  
You can have functions where you've put in your code, or maybe classes with hash functions.

2:48  
So you organise your code, but then you want to have separate files as well.

2:54  
You don't want to have your entire programme in one ginormous dot PY file.

2:59  
Why?

3:00  
Well, because it's harder to work with if you have lots of developers, because they'll all be changing the same file.

3:07  
And if you start using a software repository, that could cause you trouble.

3:11  
It's also perhaps better that you organise it with files so that you can see what each file is meant to be doing.

3:18  
You can collect things together into a file.

3:21  
Now with Java, you'll see that things go into Java files and you have to have a public class which has the same name as the Java file.

3:32  
In other languages you have other concepts.

3:34  
So for example in C or C you have header files which you include, and Fortran you have ink files.

3:41  
It it's a general concept where you want to be able to split your programme into separate files.

3:48  
Now the way Python does it is that you have modules.

3:52  
Now, confusingly, a module can be a single file, or it can be a directory of files, and we'll look at both of these.

4:02  
When it comes to the final assessment that is the exam, I'm only execting you to use a module as a single file, not as a directory O I'm including this for completeness.

4:12  
It's documented also in the Python programme examples and I'll revisit it again in 993.

4:19  
Well, I'll give you a piece of code which uses a module as a directory.

4:24  
OK, so the name of the game is really to have a better programme structure and to group those files into a module where if it's a directory, you could have several directories making up that module.

4:36  
And yes, you can have sub modules and everything else.

4:39  
OK, so here is a module as a file.

4:44  
Very simply we start off with a file and this file is called my\_code dot PY.

4:52  
Now the PEP 8 naming convention for module names is snake case as far as I'm aware.

4:58  
So the module name here is in snake case.

5:02  
Now ideally make your module name meaningful.

5:05  
The other thing to avoid is don't give your module name a name which you are then going to use as a variable because then Python will be confused.

5:14  
Are you referring to the variable or the module?

5:17  
A little bit like don't use a variable name which is a type name.

5:21  
Same idea.

5:22  
So be careful of that.

5:23  
You want to call the module name something else.

5:25  
And then inside here we've got whatever you want.

5:28  
It could be some functions or classes, doesn't matter.

5:32  
Here I've just put a little function, it's a very trivial function that's inside this file.

5:37  
And then outside I've got another Python file.

5:40  
In this case, I've called the other Python file modules dot PY.

5:43  
But that's not the important point because this one is going to import the other module.

5:48  
So what happens is at the top of the other file we say import my code.

5:53  
Notice my code is just the name of this first Python file without the dot PY.

6:00  
Here we have import my code.

6:03  
Now once we've imported my code, we can then refer to anything that's in the my code dot PY file using this notation.

6:13  
This is the module name dot and then the in this case it's the function name.

6:18  
If I wanted to use a class that is defined in this my code, I'd just have my code dot the class name and then I'd have rounded brackets and that would be me calling the constructor.

6:30  
The other nice thing that effectively this does is it gives you a namespace.

6:34  
Now if you don't know what a namespace is, you can switch off at this point.

6:37  
So in other languages there's a concept of a namespace.

6:40  
The idea is that you you can organise your classes so that potentially you could have the same name of a class, but it's inside a different namespace.

6:51  
It's a concept that shows up in other languages like C++.

6:55  
Anyway, you then can say this function belongs to that module, all right?

7:01  
And off you go, you just call it and it works as normal.

7:05  
So this my code is not a class, it's the module.

7:08  
All right, fine, so there it is, a single file.

7:13  
Now let's have a look at the directory example and then we're going to go and look at the code and see it running, right?

7:22  
One other gotcha before I go and have a look at the directory example, is the main entry point.

7:28  
Somebody asked me helpfully in the lab last week, what was I doing here with if name equals, equals main, right?

7:38  
Python, unlike other programming languages I can think of, doesn't have a main function where the programme starts.

7:45  
If you're using Java, you'll have noticed you've got a main where the programme starts from.

7:51  
C is the same, you've got a main, CC, sharp, blah blah blah.

7:56  
They all have mains, but not Python.

7:58  
Python basically will run anything that's not inside a function or a class.

8:04  
Now you've got a problem, which is that if you import a module, that is, if you're treating a file as a module and you import it, if it's sat outside like this one, it will just run that code.

8:17  
Which could mean that your programme is being run when it's imported and you don't want that to happen.

8:23  
Well, in some cases you might but mostly you don't.

8:26  
So what this little bit of syntax means is if somebody executes the Python file, then it will go inside this if clause and run the next line.

8:40  
So here it is in usage terms you say Python or Python 3 main dot PY.

8:46  
In this case what happens is it will in this case print hello and main.

8:51  
So it's executing this line and this line.

8:53  
Whereas if you import main, so importing main dot PY you can see it only executes hello.

9:01  
So the purpose of this clause if the name is equal to the main is to prevent execution of code that's outside of a function when you import it.

9:15  
OK anybody got any comments or questions about that?

9:23  
No, all right, fine.

9:25  
So that is really needed for when you're treating a file as a module.

9:32  
All right, now you could say, well, I'm going to treat my files not as modules themselves, but I'm going to create directory modules.

9:40  
I'd say still leave that in.

9:42  
If you really want some test code, don't omit it because somebody sometime might import your file as a module and then execute whatever it was you didn't want them to run.

9:55  
OK, so modules and directories.

9:59  
So in this case I've got a slightly more complicated example.

10:04  
Now here you can see I've created a directory and it's called mymodule.

10:09  
Again, it's PEP 8 namings convention, so it's snake case.

10:14  
And then inside this directory I've got a few more files.

10:18  
And then at the top level here, I've got a file called run dot PY which is going to run the things that are in here.

10:28  
OK, So most importantly, for a directory to become a module, that directory has to contain a file called\_\_init\_\_dot PY.

10:39  
If that file is in the directory, the directory can be considered as a module and you can load it.

10:46  
Now that file could be completely empty.

10:50  
All right, So if you don't want anything to happen when you import the module, you end up with an init file that's empty.

10:59  
If you want something to happen when you import the module, you put it in the init file.

11:04  
Now you might say, well, we, we're never going to do that.

11:06  
Or we will.

11:07  
Well, you might, you might want to load some data before somebody calls another function.

11:12  
So that file can be useful.

11:15  
You can also define functions in there which you can then use.

11:19  
So it's not actually going to execute something, but it's just going to import some functions.

11:23  
All right, now these others.

11:26  
So if you want to have multiple files inside your module, then you create a directory which is a module.

11:32  
And then you can just go ahead and create some files inside that directory which you've got other stuff in.

11:37  
And then you can refer to them.

11:39  
And I'm going to show you how to do that.

11:41  
Now, the last file I'm going to mention is this one\_\_main\_\_dot PY.

11:46  
Now what happens with this one is that if you execute the module, that means you say Python space minus M and then the module name, it will run this file.

11:59  
Now again, I'm not going to show you this in this module, but yes, you will see it in CS 993 next semester.

12:06  
So if you execute the module, it will run whatever's in that file.

12:10  
Same idea, same concept as this escaping that I showed you for when you're treating a module as a file.

12:18  
OK, so with the structure here, you can see that I can say from my module, meaning this one import functions.

12:27  
Now here I'm saying I want to import what is in this functions dot PY and then I can use whatever is in there.

12:35  
So in this case, I've got a function called fun one and then I can import the other one.

12:39  
You can see here and I've got in there a function called fun 2.

12:43  
So you can see that I can have a directory which actually has files in, and I can import from whatever I want inside.

12:49  
I can also say import my module.

12:52  
And what that does is it's going to run this one in it, and then I've got whatever's defined in there.

12:59  
So in this case, I defined another function in there for a bit of a laugh called fun three.

13:05  
Now, programming is a choice.

13:07  
There are lots of choices so you can decide how your module is structured.

13:12  
You could have sub directories, you have different file names, it should be meaningful, it should be sensible to you and other developers and then you can keep things nice and tidy and test them accordingly.

13:25  
OK, so does anybody have any comments or questions about this before I look at some code?

13:32  
Yes.

13:36  
OK.

13:37  
Run dot PY is this file down here.

13:39  
So it's just a little file that I wrote that shows you how to use the module.

13:45  
OK, sorry, I should have labelled it.

13:47  
This is run dot PY.

13:50  
Anybody else?

13:52  
Yeah.

14:00  
I could if I wanted to.

14:02  
Having imported the module, I could say my module dot functions dot fun one, right?

14:10  
As long as that is has been imported, you could have something that's very long referring to the other one.

14:16  
Often we don't do that, We just want to make it a little bit easier to write the code down below.

14:22  
Normally speaking, one the reference to a module before a dot is good because then you've got some kind of namespace.

14:30  
But having multiple starts to look messy.

14:33  
I mean, you can, but yeah, looks messy, especially if you've got the PEP eight line length constraint.

14:40  
I've got to import that and then dot and dot and dot and you get the idea.

14:46  
Anybody else?

14:49  
All right, let's go and look at some code, shall we?

14:53  
Where's my code?

14:54  
Here it is.

14:56  
Great.

14:56  
We'll make it a bit bigger because it's awfully small.

15:00  
OK, so that's my let's go back here.

15:05  
Let's make it a bit bigger for those people sat right at the back.

15:09  
Here is modules dot PY and let's debug it and see exactly what happens.

15:17  
We'll come down here and we'll run in debug.

15:20  
So you can, if you've got a programme that comprises one or module, more modules, as long as you start from wherever the programme starts, you can put break points in the separate files of the modules and the debugger, or just stop it wherever it is in the programme execution.

15:36  
I'm just going to put a break point here at the top so that I can then navigate down just to prove to you it's not doing anything strange or odd, it's just following the execution logic when it runs right.

15:56  
OK, we'll get rid of that.

15:58  
Oops, I'll just get rid.

16:01  
So here we are, We're at line 8.

16:04  
So we're gonna step into the function and lo and behold, we appear in my code.

16:09  
Let's hop forward and we've come out of my code and we've got a result.

16:14  
OK, very simple, just like calling a normal function, but the function has to be or in this case it is in a separate dot PY file.

16:24  
All right, so the other one, here we go.

16:28  
This is the main.

16:29  
So what we've got here is we've got a statement that's outside the hello and another statement that's inside the main.

16:36  
So what am I going to do here?

16:38  
I'm going to sit in this directory hopefully.

16:43  
Yes, let's do that.

16:45  
I'm going to open here folder, So let's say file open folder and then I'm going to go into extension 2.

16:55  
This case.

16:57  
Yes, right now the file open folder is helpful because then I'm sat in that directory and then I can refer to the module directly.

17:07  
If I didn't do this, I'd have to change the Python path.

17:10  
I'm not expecting you to change the Python path.

17:12  
And again, you can switch off.

17:13  
If you don't know what Python path is, we don't need to worry about that.

17:18  
So OK, so I want a new terminal.

17:23  
So here what I'm going to do is I'm going to say Python, oops, Python.

17:28  
And then I'm going to say import main, right?

17:34  
What happens?

17:35  
Oh, it ran hello.

17:37  
It didn't run the other piece of code.

17:40  
So if we go back here, you can see move this out of the way a bit.

17:47  
You can see what it's done.

17:48  
It's run this bit, but it hasn't run that because that's protected by the equals equals main.

17:56  
So if I bring this back, we can now get out of that like this.

18:01  
And instead we could say Python main dot PY and you can see it executes both pieces, the hello and the main.

18:11  
All right?

18:11  
So that that is what's going on with this.

18:17  
If name equals equals main.

18:19  
All right, so let's go to the next bit.

18:22  
I'm gonna hop into the other folder.

18:26  
All right, so modules as a directory.

18:41  
So somebody asked me what is run dot PY it's here.

18:44  
You can test it and play So in this case, we're gonna import some things.

18:50  
Actually, let's put a break point in here.

18:53  
Yeah, why not?

18:54  
We'll put a break point in this in it just to have a bit of a laugh and yeah, maybe we'll do that in a minute.

19:04  
OK, so let's run and see what happens.

19:23  
Arrive at line 2.

19:28  
All right, so this happens already because we said from.

19:33  
All right, so from that it's actually imported this already.

19:36  
Let's go on.

19:38  
Yes, fine, we'll carry on.

19:40  
So we're going to go back to from because we said from again, right?

19:45  
And then it's finished.

19:46  
Let's put this down here instead.

19:55  
We're just going to test one of these out and then we'll go on to the next idea.

20:02  
OK, so that's me importing the module.

20:04  
So you can see we've come into init dot PY and let's go on.

20:11  
Let's get rid of that.

20:12  
So here's our function fund one.

20:15  
Now you remember this fund one is implemented in functions dot PY in here, right?

20:21  
So let's go and see where we arrive.

20:24  
Oh, look, here we are.

20:25  
We're in functions dot PY.

20:27  
All right, so nothing particularly strange is happening.

20:29  
We're just using a function that happens to be in a file, which is inside a directory.

20:34  
Whatever.

20:36  
OK, let's stop that now.

20:38  
I told you that if you execute the module, then you will run the main.

20:46  
So let's see that.

20:48  
So here's the main.

20:49  
So you see, it says run something.

20:52  
Not a very useful thing to do, but whatever.

20:55  
This, by the way, is not inside an if name equals equals main.

20:59  
It doesn't need to be because if your directory, sorry, if you are using a directory as a module, you don't need that.

21:06  
So let's see.

21:07  
And nobody's going to import that and not expect it to run something.

21:12  
So let's put in here a terminal and we're going to say.

21:20  
Now, the way you run modules, you say Python minus M, and you can get Visual Studio Code to run a module as well in debug mode.

21:27  
But anyway, Python minus M and this one's called my module and there you go.

21:36  
It printed out.

21:37  
Run something.

21:38  
So it has Oh yes, and you can see it's also it's executed this because it imported the my module that was coming from the import dot, sorry, the init dot PY, and this one is coming from the main dot PY.

21:53  
OK, so play around with that code.

21:56  
Just sort of demystify it because nothing really is complicated here.

21:59  
We'll just put it in files.

22:01  
Anybody else got any questions before I move on to unit tests?

22:08  
No.

22:08  
OK, fine.

22:09  
Let's carry on then.

22:11  
Let's go back to the lecture notes.

22:13  
Sorry.

22:15  
OK, Unit tests, What are they?

22:17  
Why do we wanna use them?

22:19  
Now?

22:19  
You will hopefully see or you have seen unit tests in Java.

22:24  
I'm gonna show you unit tests in Python.

22:27  
Pretty much every programming language that's useful has a unit testing way of setting it up.

22:32  
There are programming languages that don't.

22:35  
Yeah, you can ask me about that later, but ideally you're using a programming language which does allow unit tests.

22:43  
What's the purpose of these?

22:45  
The unit tests are there to verify that a unit, which could be a function or a class or whatever you want to call a unit, behaves as you expect.

22:55  
Now normally this is a simple call where you pass the function what you expect, and you might pass the function what you don't expect, and you check to see the function behaves as it should.

23:07  
So does it return the right answer or does it return the right error when you give it the wrong input?

23:13  
Now, why do you want to do this?

23:15  
Well, if you're coding on your own, you know, you're in your own shed at home.

23:19  
You're bashing away from Friday until Monday morning, just you and the computer, nobody else involved.

23:24  
You can be very, very careful and your code has no bugs, and I've done that myself.

23:30  
It's OK when it's just you.

23:33  
However, you may be coding away and even like six months later, even if it's just you, you accidentally break something somewhere else in the code by a change and you didn't realise that.

23:45  
Now once you put in lots more developers like, I don't know, three or four other people and they keep changing your code and they don't remember everything that came through your head in that one glorious weekend when you built the whole thing in the 1st place.

23:58  
They will likely break something.

24:00  
So the idea is that the unit tests are there to establish that the code is still OK.

24:07  
They're a bit like a smoke test.

24:08  
They don't test everything.

24:10  
They just test to see, yeah, does it work as it should do, yes or no?

24:16  
Now, if you're going to release code, you should then run a bunch of other tests, use your acceptance tests, system tests.

24:22  
We'll discuss it next semester where you verify does the code really, really work in all of the weird scenarios.

24:29  
So unit tests are lightweight tests that just check does it work, yes or no?

24:35  
OK.

24:36  
The other good thing is they can be automatically run when you take some software and you put it into a repository.

24:41  
And again, we'll cover this in nine O 3 next semester.

24:44  
The idea is you send your code into the repository and it can run the unit test for you and then say if they're broken or not.

24:52  
So the software's only accepted as a change when they don't break.

24:57  
Now, it could be that your software's fine, but you forgot to change the unit tests, in which case you're still going to have to fix that, and whoever's controlling the repository will tell you to fix it.

25:07  
All right, How do you implement them?

25:09  
Now, unit tests should be in separate files.

25:13  
Why?

25:14  
Well, because they're not actually part of the production code.

25:17  
They're not part of the thing that you should be running when you've deployed it.

25:21  
So imagine you run a sorry, write a nice piece of software and you sell it to somebody and you leave the unit tests in there.

25:28  
And then the unit tests are somehow accidentally executed and the software crashes.

25:33  
If you leave them in there, they may have vulnerabilities and somebody might manage to execute them.

25:39  
So they really shouldn't be in there.

25:42  
OK, So they're normally not distributed with the software and they're normally in the same place as the software when you're working as a developer.

25:51  
So you've got one repository which has inside their unit test and the rest of the software you normally separate them into separate directories.

25:59  
So you might have a tests folder and the rest of the software is in another day structure.

26:04  
Now for our assessment, the final programme exam, I don't need you to do that.

26:10  
You can put the unit test in one file and the other code in another file.

26:13  
And I explicitly tell you to do this, but again, in 993, you'll see it where the test is separated into a separate directory, which is what you normally do for a piece of production software.

26:24  
OK, you should make the unit test names as obvious as possible.

26:29  
So don't give it a name like test ABC when it's actually testing a separate function with a different name, because then when it crashes or fails, it'll print out the error of this test with this name failed.

26:41  
And if it's called ABC, nobody's got a clue what it means.

26:46  
You can configure the software before the test, right?

26:49  
You need to remember what steps have to be done.

26:52  
So maybe some data have to be loaded into memory.

26:55  
Maybe you need to do a bit of processing beforehand.

26:57  
And then the other key thing about unit tests is that you define what should happen, right?

27:03  
So if you call this function, it should give back this value.

27:08  
Now when you're building unit tests, you want to use simple values that you can compute with your brain, right?

27:15  
So if you going to put in, I do not know, two or three, and you get back another value, great.

27:20  
You put in there like 345.62 and you go, yeah, it should be.

27:26  
This is the answer, shouldn't it?

27:27  
Calculator, It is just going to slow you down South.

27:30  
Pick some values that are obvious and then you can quickly verify what the result is and if it's OK or not.

27:39  
Right.

27:39  
So I'm now going to show you some unit tests.

27:42  
Does anybody want to ask me any questions about unit testing in general?

27:48  
No.

27:49  
OK, so let's go back to where we were.

27:55  
We'll open the folder where we want to be, file open folder, and we'll go into our unit test.

28:03  
Example 4 here.

28:10  
OK, so this is basically the same sort of thing that you'll get in terms of structure as in the programme exam, you've got one file which is the module, and the other file corresponds to the unit test for testing the module.

28:24  
All right, so here we've got a module and it's called example dot PY.

28:29  
So I've decided to call the file that's going to test the module test example dot PY.

28:34  
Pretty self-explanatory.

28:35  
It's going to test whatever is in the module that's called example dot PY.

28:40  
Now to make this very simple, just so you can see the concept, I've only got 1 function in this module, example dot PY.

28:46  
There could be a whole bunch of things.

28:47  
You could have two classes in there, a bunch of functions, doesn't matter.

28:51  
So in here we have a function, it's called double.

28:55  
And this function, all it does is it multiplies the number by two and returns it.

29:00  
I've only written this just to show you the idea.

29:02  
You could have a very complicated function doing very complicated things, including throwing an exception.

29:08  
And yes, you can test if it throws an exception.

29:12  
So here we go, the unit test example.

29:14  
So first of all, I import the module example just as we were previously importing modules.

29:22  
And then I import another module here called unit test.

29:25  
I'm importing this one because I'm going to inherit from it.

29:29  
So here we've got a class.

29:31  
Our class here is called test example.

29:34  
It should be prefixed with tests.

29:36  
So it's test something.

29:37  
In this case, I've called it test example because it's testing what's in the example dot PY.

29:43  
I'm inheriting from this thing test case, which is actually inside the module unit test.

29:48  
Now, this thing test case is clever.

29:52  
I won't go into the insurance and outs of how it does it.

29:54  
You can ask me later and I'll tell you if you really want to know.

29:56  
But what it does is that when we run the main that's in this test case, it will automatically pick up any function here that's prefixed with test.

30:07  
All right, So all you need to do is make a bunch of functions inside here which are member functions which are called test\_something and it will automatically pick them up as unit tests.

30:17  
You don't need to go.

30:18  
I've got to call that function.

30:20  
It will just do it now by default, I think it will order them alphabetically and execute them.

30:27  
So you can actually use the order, right?

30:32  
What can you see inside test double here?

30:35  
Notice I've called it test double because it is testing the double function.

30:40  
I have one check, so there are several of them where you can say self assert equal or self assert true or self assert, whatever you want.

30:51  
If you use Visual Studio Code, it will suggest a bunch of these to you.

30:57  
These functions, assert equal or whatever are actually coming from this class test case.

31:04  
All right, so we're not defining these functions, we're just using them.

31:08  
And then in this case, what we do is we give it what we want in terms of we are going to call the function and what we expect.

31:18  
So I'm going to call this function.

31:19  
I expect it to have the output 4.

31:22  
Now, if I hovered over again, you can see that we can also put a message in.

31:26  
So when it crashes, you could use this third input parameter to provide a message.

31:33  
So if we run this, we can just run the file and it runs and lo and behold, it says, OK, run one test.

31:44  
Now there's only one test because there's one function called test double.

31:48  
If I had another function called def test\_another function, it would run 2 tests.

31:55  
And you can see, yeah, it passes because yes, 2 \* 2 is actually 4.

32:00  
Now if I make this five and we run it again, this time it says failure and it tells us where it failed.

32:11  
So it says, oh, it failed 4, which is what this returned is not equal to 5, which is what I wanted.

32:20  
And it tells us it line 13, this failed, which is quite helpful, which is in actually in this test double here.

32:28  
So that's why we want to keep the names reasonable, because as soon as I see that I know exactly where the problem lies, I can go and debug it.

32:38  
So let's put that back so I don't confuse myself later.

32:42  
Does anybody have any questions or comments about that?

32:45  
Yeah.

32:57  
How long it takes to run the code down here?

33:01  
Yeah.

33:02  
So this is just coming from the unit test framework.

33:08  
There are two ways of running unit tests in Python.

33:10  
You can execute it like this, which is using this unit test.

33:14  
Well, there's another idea which is this.

33:17  
It's called π test.

33:19  
I believe both will tell you how long it takes.

33:22  
So this is actually coming from the code that's in this module unit test.

33:27  
I didn't have to write it.

33:29  
The problem with unit tests is that if they take a very long time and they automatically run when you push code into the repository, at some point the build that happens will time out.

33:41  
So you want your unit test to be short, otherwise it will cause you problems when you put it into the repository.

33:47  
So one or two seconds is fine for all of them to run.

33:50  
If it's taking minutes, then you're going to have a special, you're going to have to have a special build pipeline to go with your registry repository to do that.

34:00  
Now, you might actually use the unit test framework to run more complicated tests, which could take minutes, but you're not gonna run those automatically in the same way as these.

34:10  
All right?

34:10  
So keep an eye on the time, total time, maybe a few seconds for all your unit tests.

34:15  
OK, maybe up to a minute.

34:17  
Anybody else?

34:21  
OK, fine, let's go and have a look at the other one.

34:28  
Open folder Unit Test 5.

34:38  
So here we've got a more complicated function.

34:43  
So what am I doing here?

34:44  
I've got a function called swap and this function is, sorry, potentially raising a type error.

34:52  
So type error is a kind of exception.

34:55  
So here I am checking to see if the type of values is a list and if it isn't I am going to go there type error basically stop the programme, throw the type error.

35:07  
So it won't go to line 8.

35:09  
In this case, the raising of the exception immediately stops the programme and we can run it down here.

35:17  
So let's see if we run this.

35:19  
This runs fine.

35:21  
Yeah, it's swapping the list.

35:25  
You can see here my original list is 1, 2.

35:29  
And then if I run the swap function, it swaps around.

35:33  
So it's now 2, one.

35:34  
All right, so it's taking a list and it's just swapping the order of the list.

35:40  
Great.

35:41  
Let's have a look at the test.

35:44  
So here we've got a normal test, so test swap.

35:48  
So I'll just get rid of this so you can see more of the code where I have put in a list and we expect that that swap function will work as we think it should, meaning that we've got the input list and our expected list is just swapped around.

36:04  
So this is the normal use.

36:06  
All right.

36:07  
So you should test how does it behave or how should it behave normally.

36:11  
Now I've then got two other things that somebody could do which could break my code.

36:17  
So I'm just checking that it doesn't break the code.

36:20  
So if the person provides a list that's empty, I expect this code should raise an index error.

36:30  
Now it will do that I haven't raised an index error myself, but the code will because I'll be looking for an index that doesn't exist so I can check if an index error is raised.

36:42  
So the way you check if an exception is raised, excuse me, is that you have to call this function, assert raises, you then give it the exception type.

36:52  
Then you give the function that you want to call and what you're going to give to the function.

36:56  
All right, So the way you call it is a little bit different to this, but again, it's like cooking.

37:01  
You just follow the recipe and hey presto, you have a cake.

37:05  
In this case, I'm checking to see if the type error is raised.

37:11  
I've done something rather naughty here.

37:12  
I've given it an integer and not a list.

37:15  
So I've said ha, ha, have an integer.

37:18  
Enjoy that.

37:19  
So you'll see that this will run fine because each one of these will pass as in the exception is raised.

37:27  
So if we run this, yeah, OK, fine.

37:31  
So what that's doing is it's checking to see get rid of this.

37:36  
It's checking to see if all of these requirements are all right.

37:40  
And if one of them wasn't OK, then it would fail.

37:44  
So for example, if I make this into a square brackets here, so this is now going to be an empty list again, the function will raise an index error.

37:55  
It won't raise a type error when it is an empty list.

37:58  
So we can run this again and you can see that it fails and it fails with the index error, which is not what I said it should fail with.

38:12  
I said it should be a type error.

38:13  
So let's put that back.

38:15  
So what are we demonstrating here?

38:18  
We're demonstrating that yes, you should test the normal use and yes, you should test the obvious failure.

38:25  
Now, you cannot test everything.

38:27  
There could be some crazy idea that somebody had that you didn't realise could exist.

38:32  
And sometimes what happens is when somebody raises a software defect or a bug and says, oh, your software is crashing, you then think, oh, we've we'll fix that and we better put a unit test in to catch it.

38:43  
So don't spend vast amounts of time thinking about other unit tests.

38:49  
Do a little bit of work.

38:51  
And then if some bug is reached and you need to patch it, then you might want to put another one in basically.

38:57  
OK Anybody got any other questions or comments?

39:01  
Yes, yes.

39:14  
Yes, yes, definitely a good idea.

39:16  
I mean I could have a separate function here.

39:18  
This could be test swap\_normal and then I could have test swap\_empty list or something like that.

39:26  
Ideally, with functions, you should make them simple enough that one could pass and another might fail or so on and so forth.

39:35  
I've put it all in one here.

39:37  
Just this kind of demonstration.

39:38  
This is fairly short, but if I had more complicated code, like I was having to do a lot of work for the normal case, I'd probably just put that in a\_normal, and then another one\_whatever.

39:49  
Yeah.

39:50  
See, I mean, it's like administration.

39:53  
You want to keep all your paperwork nicely filed.

39:55  
So if things start to become messy, then, you know, separate them out.

40:01  
Anyone else?

40:03  
Yes.

40:08  
So the format here is that you are creating a class which starts with test as capital T It is inheriting from this thing test case which is actually defined in there.

40:23  
And then any functions you want it to run as unit tests have to be prefixed with test\_ so.

40:29  
If I CREATE another function called run it, won't run it there.

40:33  
Are other functions if you look in the documentation that you're interested there.

40:37  
Are other functions in unit test that will do things when you open the class or their setup or setup class functions we.

40:47  
Don't need to use those for this module if.

40:49  
You want to learn about them go, for it they.

40:51  
Are potentially useful.

40:53  
I think I introduced them again next semester, but I can't remember for sure.

40:57  
So if you come across an exam question, I'm not needing you to have any setup or setup class functions in there, just simple functions like this.

41:08  
OK, so we're introducing the idea.

41:11  
Anyone else?

41:14  
Yes, yes, Like that.

41:26  
Yes.

41:29  
Yeah, I right.

41:32  
OK.

41:32  
This is coming back to the kind of simple tests.

41:35  
Yeah, you could have a list that's very long.

41:38  
Again, you probably want to make the unit test somewhat simple, so you'd probably go for a short list.

41:44  
I mean, if I was worried that I wasn't sure what would happen if I had three elements, right?

41:50  
Is it really going to or maybe 4 because like does it really swap or does it just leave the middle?

41:55  
I could go, let's have a four element list, but I probably wouldn't become more complicated than that for this function because there's no point in terms of testing it.

42:05  
So yeah, you need to consider what is the sort of minimal size of input data that's useful for testing the function.

42:12  
And like I said earlier, a unit test isn't the only thing you're going to run when you deploy or sorry, before you release the code, you might come up with a nasty example where you're going to put in a very, very big list and you want to see does the programme work?

42:29  
Does it start using vast amounts of memory?

42:30  
Does it crash?

42:32  
Which again is a is a different type of test for normally for unit tests, you keep the the data that you're dealing with small.

42:41  
One of the concepts we saw last week was that of input files.

42:46  
Now, instead of using an input file in a unit test, you can use a string.

42:52  
So you can create a string, and then you can create from the string a string IO, and then you can deal with the string IO as if it's an open file.

43:02  
So again, you can sort of separate yourself from having a file so your unit test can run fast with the string.

43:08  
It's a concept I'll introduce to you in a later example, but essentially you want to keep it simple and small.

43:15  
Based on what we were discussing earlier, we want this thing to run fairly quickly.

43:20  
OK Anybody else?

43:25  
Any questions at all about module?

43:30  
Anything else?

43:31  
No, it's very particular.

43:38  
If you look at the, if you look at the previous exam paper, you'll see it tells you exactly what you should do.

43:47  
The exam's written, so hopefully there are no ambiguities so we can have a look at the previous one here.

43:58  
So we go down here and it says a little bit further.

44:06  
Oh yeah, too far, Sorry.

44:08  
Right here it says create a file named test file catalogue dot PY and it contains unit tests for these two.

44:16  
And it says these tests should verify the correct behaviour of and then gives you a list of functions.

44:21  
All right, notice correct behaviour of.

44:24  
This is a big hint that I'm only wanting you to test the normal behaviour of these functions, OK?

44:31  
If there's another function in that particular problem, and I haven't listed it here, I don't need you to give me the unit test.

44:38  
OK?

44:39  
Right.

44:40  
So there, in this case, there are 5, so it's like 3 marks apiece.

44:45  
Anybody else?

44:48  
Yeah, yeah, yeah.

45:08  
OK, so explicitly what I've said here is should verify the correct behaviour.

45:14  
So I'm only looking for a unit test that's got a normal input, right?

45:20  
So I'm not looking for complete coverage of all exceptions or anything like that.

45:25  
It's just give a normal input.

45:27  
So what's going to happen here is if I'm unit testing an repr function, I want to check can I use it with eval, right?

45:35  
So I'm going to call the constructor and then I'm going to turn it into text string and then I'm going to check can I use it with a vowel?

45:41  
Does that work?

45:43  
Same for this one, this one here.

45:46  
What I'm going to do is I'm going to have a CSV file and then I am going to try to call those functions and see does it fill the class with the data that I think should be there, yes or no?

45:59  
And I can check the data values have been loaded in this one here, the find name function, I'm going to have to create a class object.

46:08  
You go back and look at the paper, it makes more sense.

46:09  
You're going to have to create a class directory that is directory object.

46:14  
And then I'm going to have to create some file inside it and then call this and check I have back the file that I think I should have.

46:24  
So the unit test for this should be relatively simple.

46:27  
They're just a few lines of Python.

46:29  
I'm not going to check the coverage.

46:31  
Coverage is a good question and something again that will be covered in 993.

46:36  
We can ask in Python how much of the code is executed by the unit test, and we'll look at it next year here.

46:44  
It's very clear.

46:45  
I just want you to create a unit test that cheques the normal input.

46:49  
As I've said here, correct behaviour.

46:51  
OK, so I'm limiting it so it's not going to take you very long.

46:56  
Anyone else?

47:01  
OK.

47:02  
All right then.

47:02  
Well, I'm gonna stop there.

47:04  
Yes.

47:07  
Yes.

47:11  
Not immediately, because we are one or two people who have got extra time.

47:16  
So I will wait until I've got them all.

47:19  
I'll double check something silly didn't happen, like maybe my place has got some error, and then later in the day I'll release them, but I they won't be released immediately.

47:28  
Basically, one, because of the extra time and two, because I just want to check there's something silly hasn't happened with my place.

47:36  
All right, Yeah, just come in.

47:40  
Yeah.

47:41  
So we'll use the week 11 lab.

47:44  
So you come in, do the test.

47:46  
When you've done the test, just quiet, leave the room.

47:50  
You know, when the time's up, you'll read the instructions on the My Place page and I'll, I'll tell you what's going to happen when you arrive tomorrow.

47:58  
But basically show up at 1 normally and, and find a seat in the lab, you know, log in, be all ready, nice and calm and everything else.

48:07  
And then I'll just tell you how it works.

48:09  
And when you're finished, just you know, you can leave.

48:11  
You don't need to stay for the rest of the time once the test is finished, OK?

48:24  
I can go and check it beforehand earlier in the day, and if it's really cold, I'm sure I can bring some other heaters if if that's a problem.

48:34  
All right.

48:35  
Yep, Yep, Yep.

48:46  
You can bring blank paper.

48:48  
That's absolutely fine.

48:49  
Bring a pad of paper.

48:50  
Totally fine.

48:51  
No problem.

48:53  
Yeah.

48:55  
Yeah.

48:56  
So bring your blank paper.

48:59  
I could probably provide some extra.

49:01  
I'll tell you what I'll do is I'll go past the the office and I'll see if I can grab some paper.

49:08  
And then you don't even need to worry about paper.

49:10  
Just bring your pen or whatever you want.

49:13  
None of the questions should require any working.

49:15  
But yeah, you can scribble on bits of papers you want.

49:19  
Anyone else?