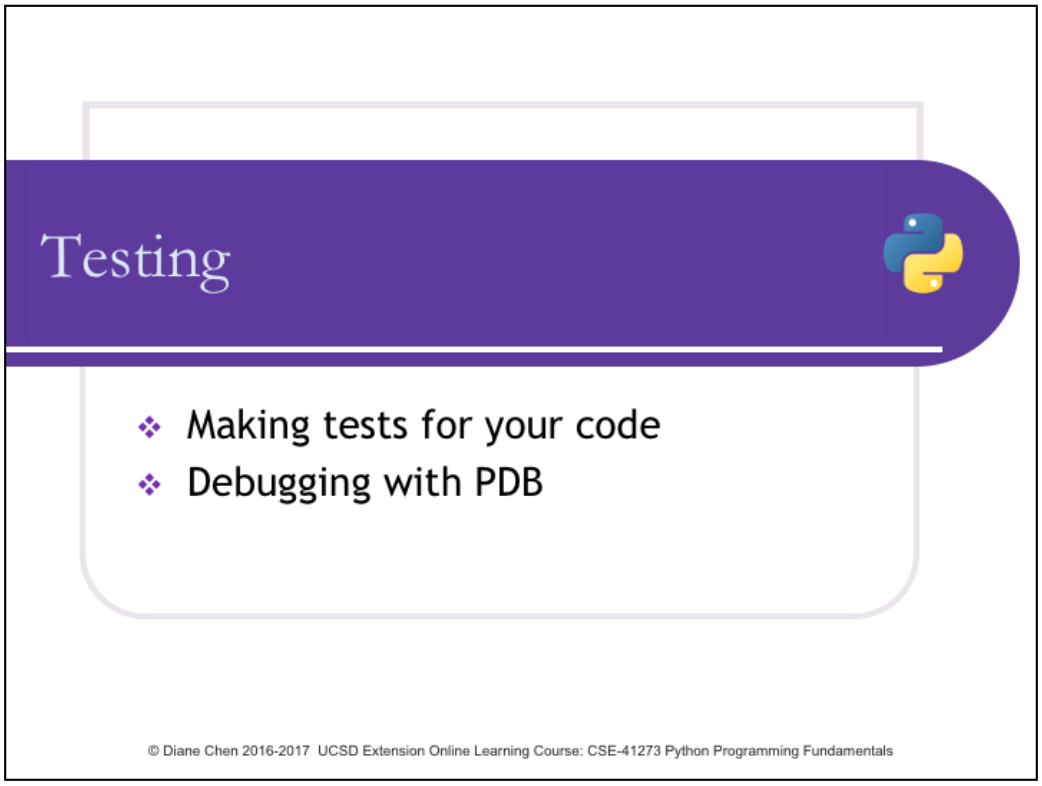
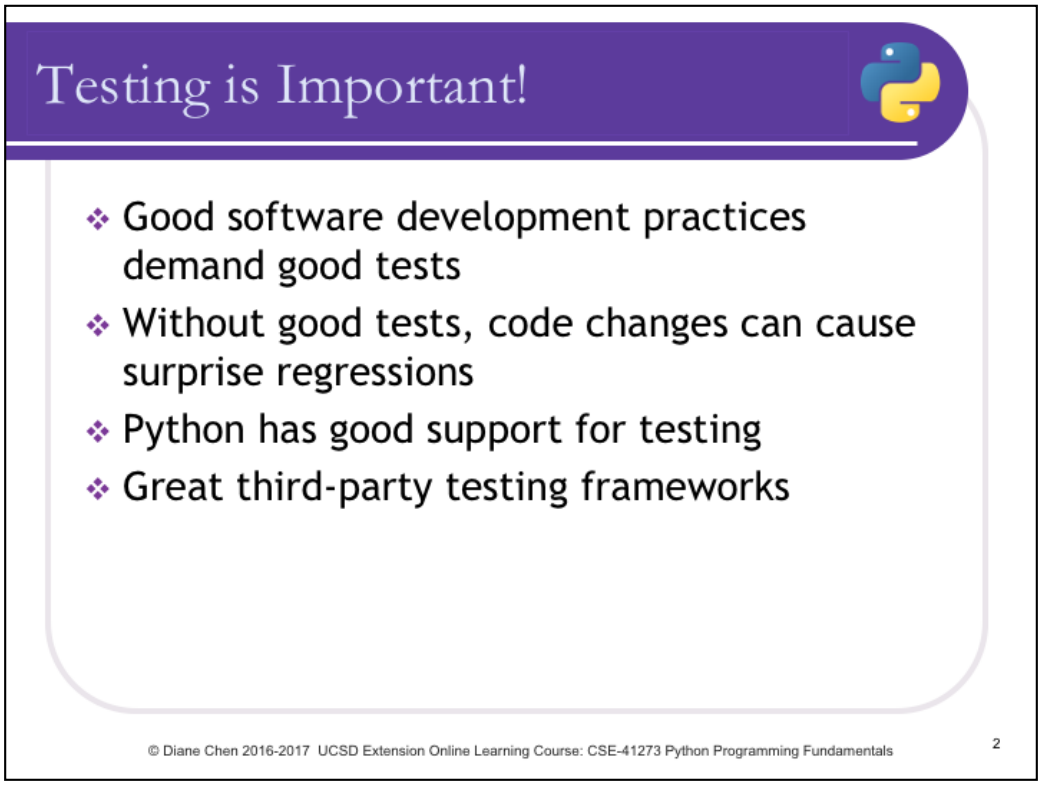
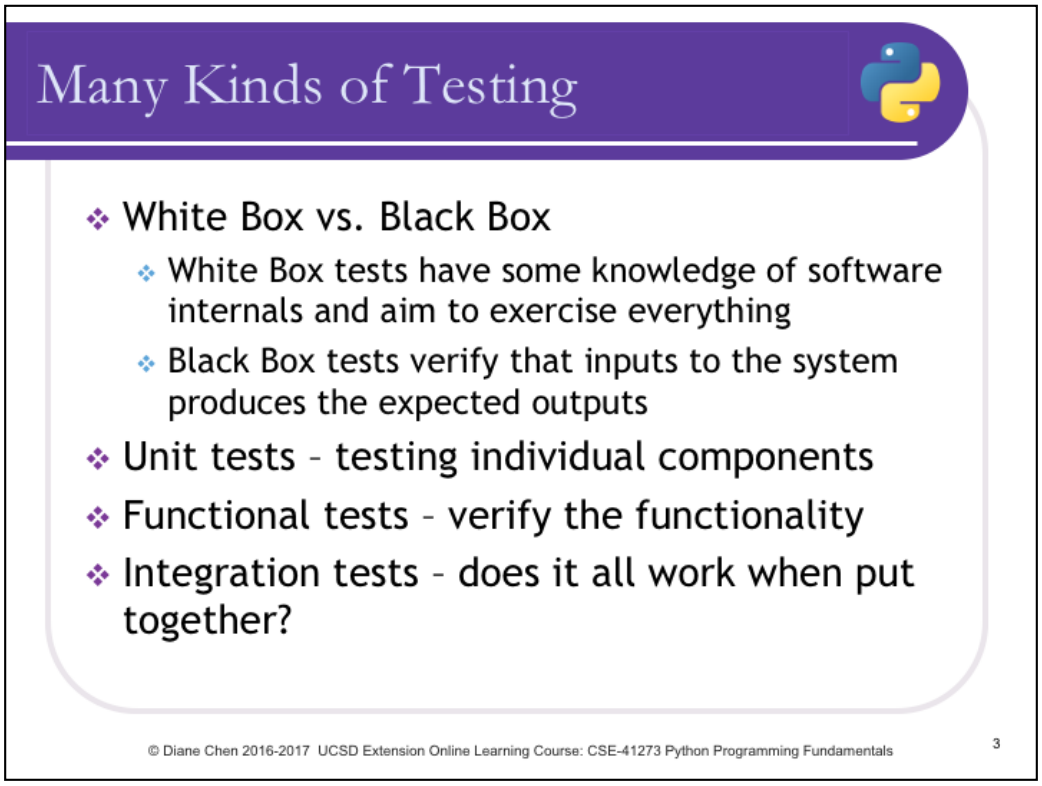


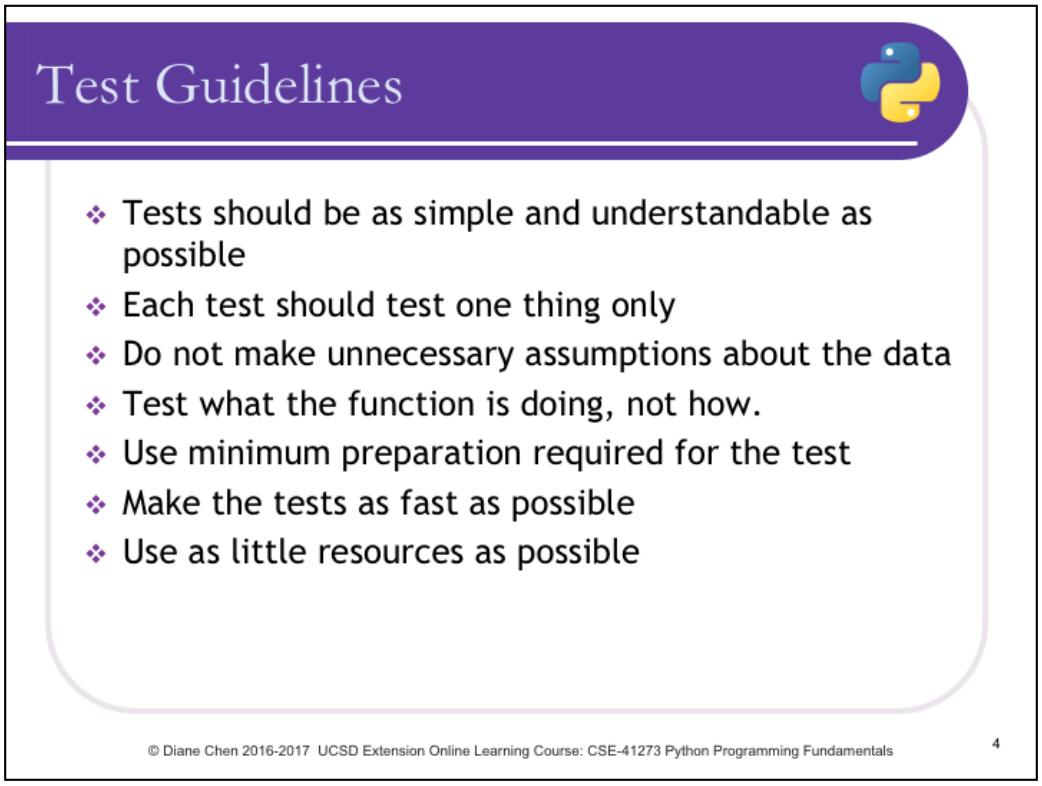
This week we are going to talk about making tests for your code. I'm also going to introduce you briefly to the python debugger, PDB, and the principles behind Test-Driven Development.



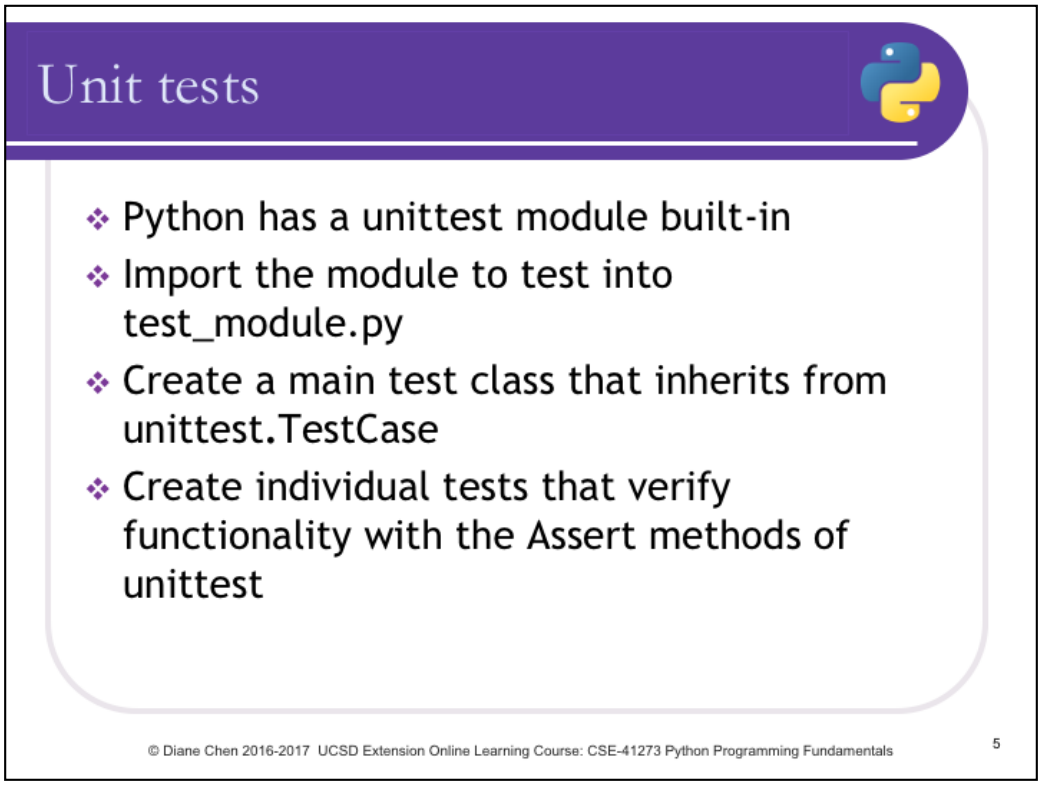
Testing is important! Good software development practices require good tests. If you don't have good tests for your code, even minor changes can cause surprise regression errors in the software. Well, I suppose all bugs are a surprise, but they are never good surprises! Python has good support for testing; it has a testing library that comes with it and there are several other frameworks that you can use for testing. They are all very similar.



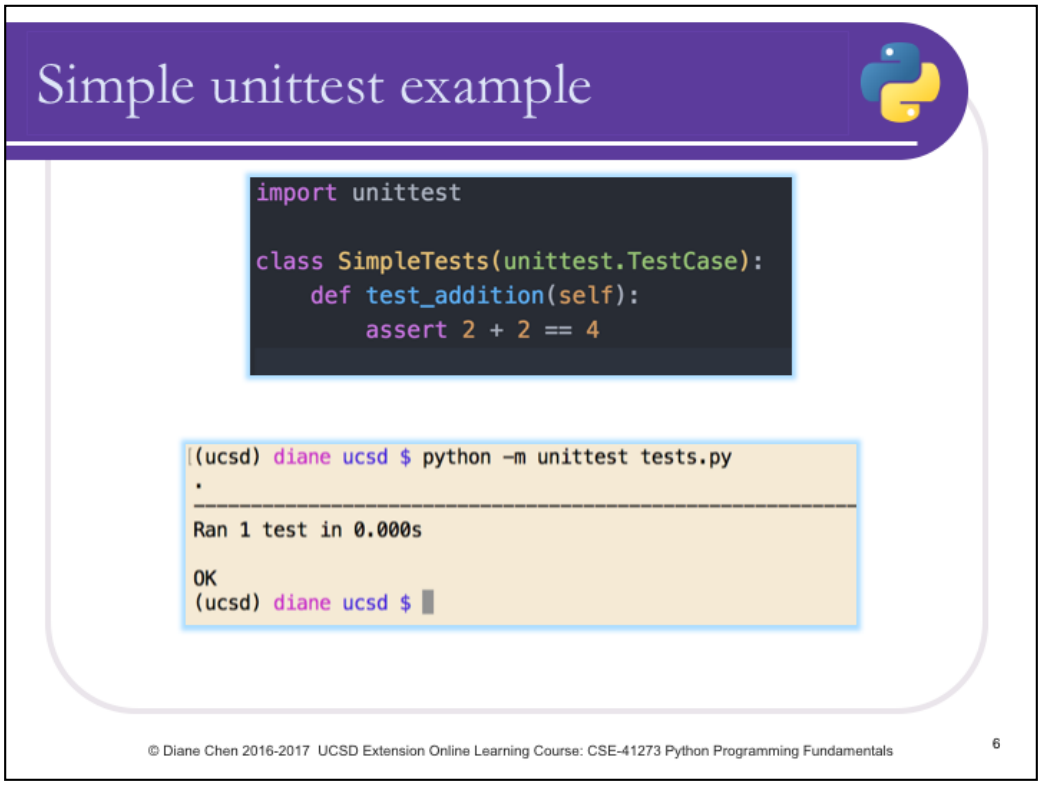
There are many kinds of testing. You will hear lots of names of types of tests – some of which are very similar and some are not. You often hear White box vs. black box testing. White box testing is where the tests are developed with some knowledge of the internals of the code; the purpose being to try to test all the code paths. Black box testing is where the tests are put together to test the externals of the program, to verify that certain inputs to the system produce the expected outputs. Unit tests are white box testing; they aim to test individual components of the software. They test each function with various inputs and ensure that each function returns the values it should. Functional tests verify the functionality of the software from a user's point of view. Integration tests make sure that all the components of the software work well together. Some component might have the correct functionality by itself, but not work properly with another component of the system. That's what integration tests are for.



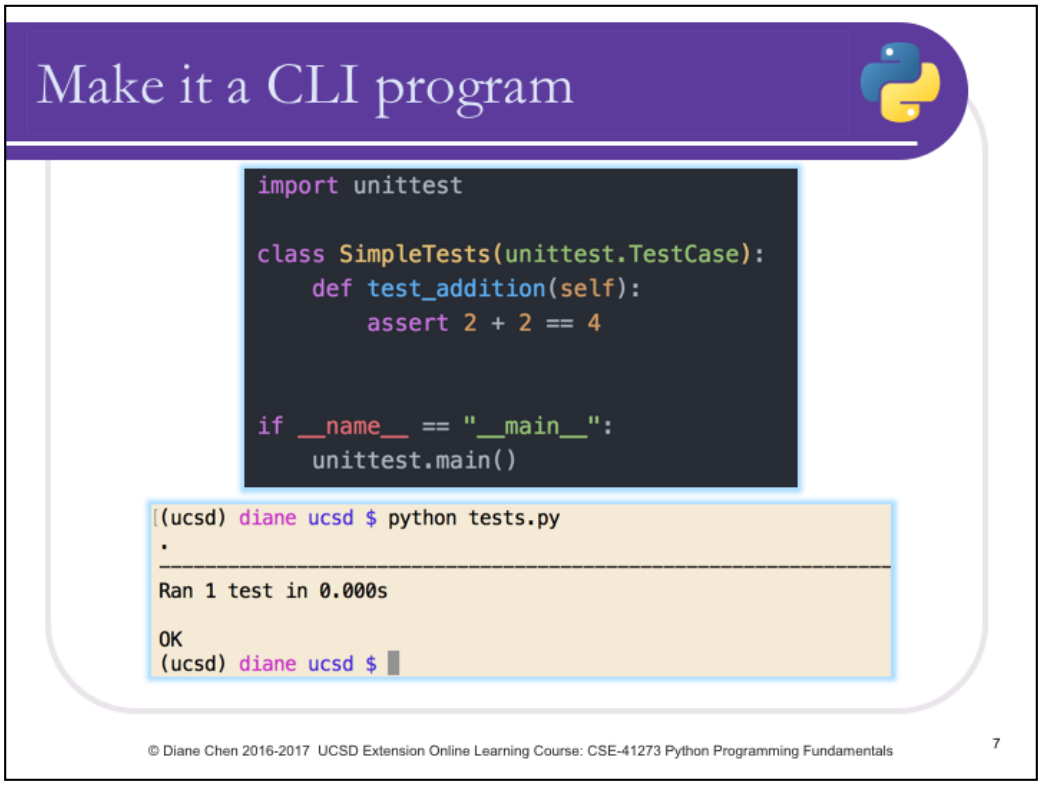
Here are some basic testing guidelines. The tests should be as simple and understandable as possible. Each test should test only one thing. IF a test tests multiple things, then if the first one breaks, you don't know whether the second thing is working or broken. Do not make unecessary assumptions about the data to be used. Always test what the function is doing, not how it does it. For testing purposes, we don't care how a function operates as long as it gives us the correct results. Use minimum preparation for the tests. It is common to have to set up a scenario of data or whatever before running a test. Make it the absolute minimum required to test the function correctly. The reason for this is we want the tests to run as fast as possible. If our tests are really slow, then what happens? People don't test as often because it is annoying. This leads to bugs possibly propagating in the software for longer than they should. As a corollary to that, we don't want the tests to use a lot of resources like memory because this slows them down also.



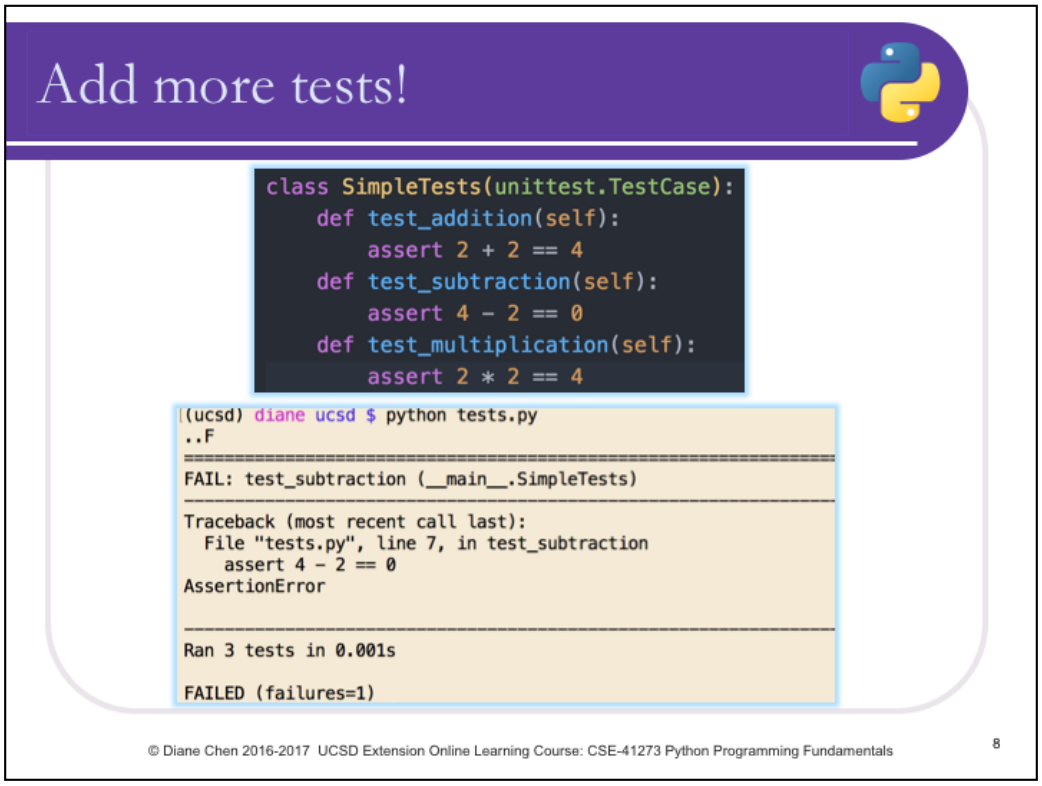
The simplest way to make tests is with the testing module unittest that is built in to Python. To make a test program, import unittest into the file, for example test_module.py. You will create a main test case class that inherits from unittest.TestCase. Then you create individual tests that veryify the functionality using a variety of Assert methods defined by unittest. We will see an example of this.



For our example, we'll make a really simple unittest in a file called tests.py. We import unittest and create the class SimpleTests that inherits from unittest.TestCase. Then we add a simple test to test addition. We can run it from the command line, python -m unittest tests.py. The –m option to Python tells it to run the module unittest with the input tests.py. Because our SimpleTests class inherits from unittest.TestCase, unittest automatically looks for methods under this class with names that start with test_ and runs them. The output isn't very exciting but that's OK! We don't want excitement in our test output. The dot means that one test passed. Then it prints a separator line and gives us the summary result. But that's a lot to type every time we want to run the tests.py test file. Let's do something about that.



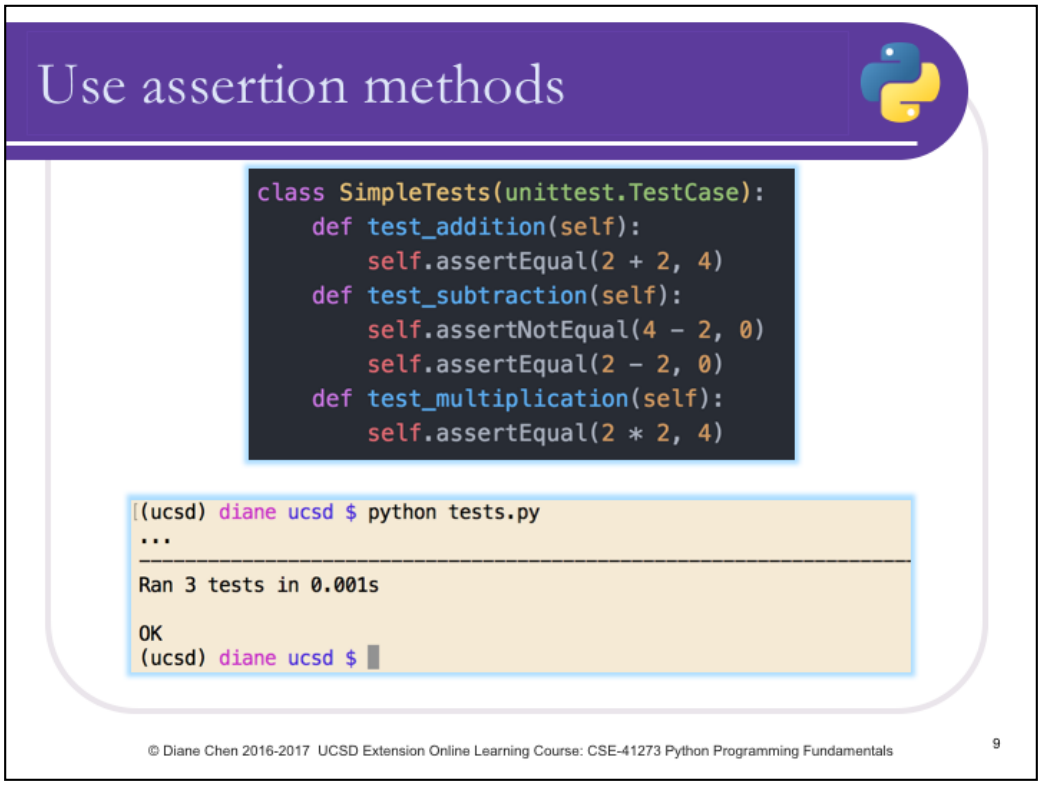
To turn it into a CLI program, we add the magic incantation for command line testing at the end of tests.py. If dunder name is equal to dunder main, then it runs unittest.main(), which is an entry point into the unittest module. Unittest runs whatever tests it finds in our file. This is much easier to type – it's just like any other command line program.



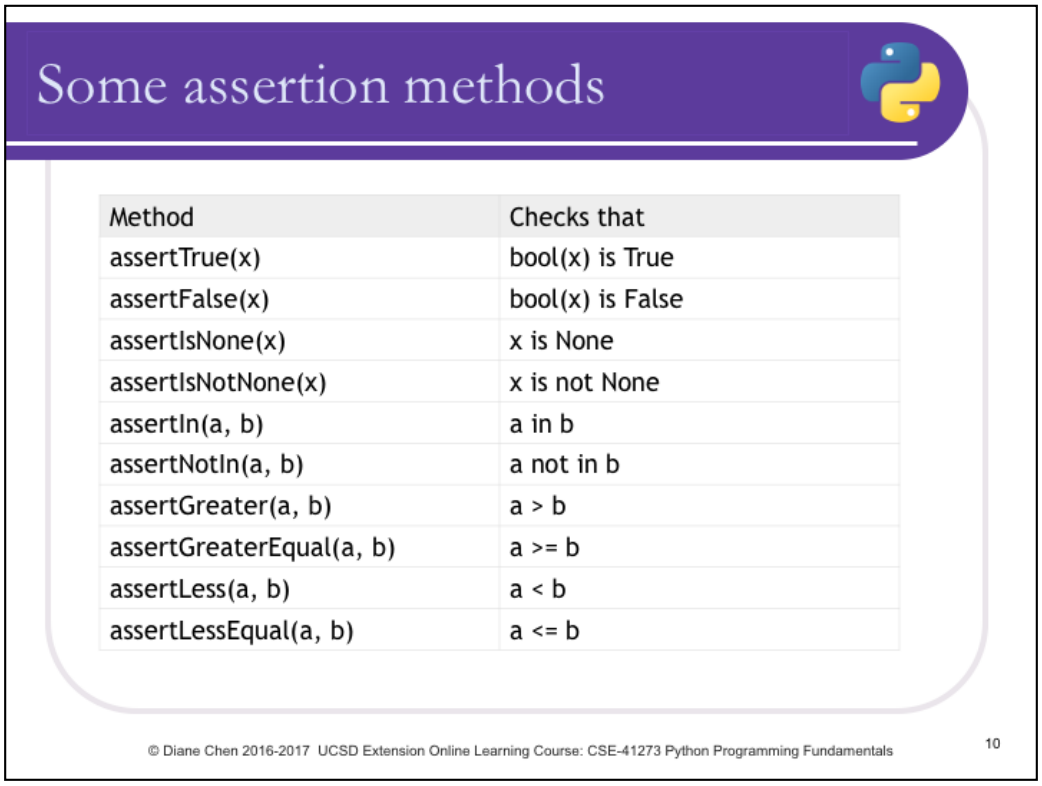
We can add more tests to the program and unittest will automatically run all of them whenever it is called. So here are a few things to point out. First of all, I made the subtraction test wrong, so it fails the assertion. You can see what the output looks like when a test fails. Instead of a dot, it displays a capital F for Failure. Then it gives some information about the failure after the summary line.

This brings up another point, that when you're making your tests, you have to be really careful to make sure you are testing the correct thing. The problem here is not in the subtraction, but in the test itself.

Also note that the tests are not necessarily run in the order that they appear in the file. All tests should be completely independent anyway, so it doesn't matter, I'm just reminding you of the fact. And, if there is an error, the failing test is identified in the output.



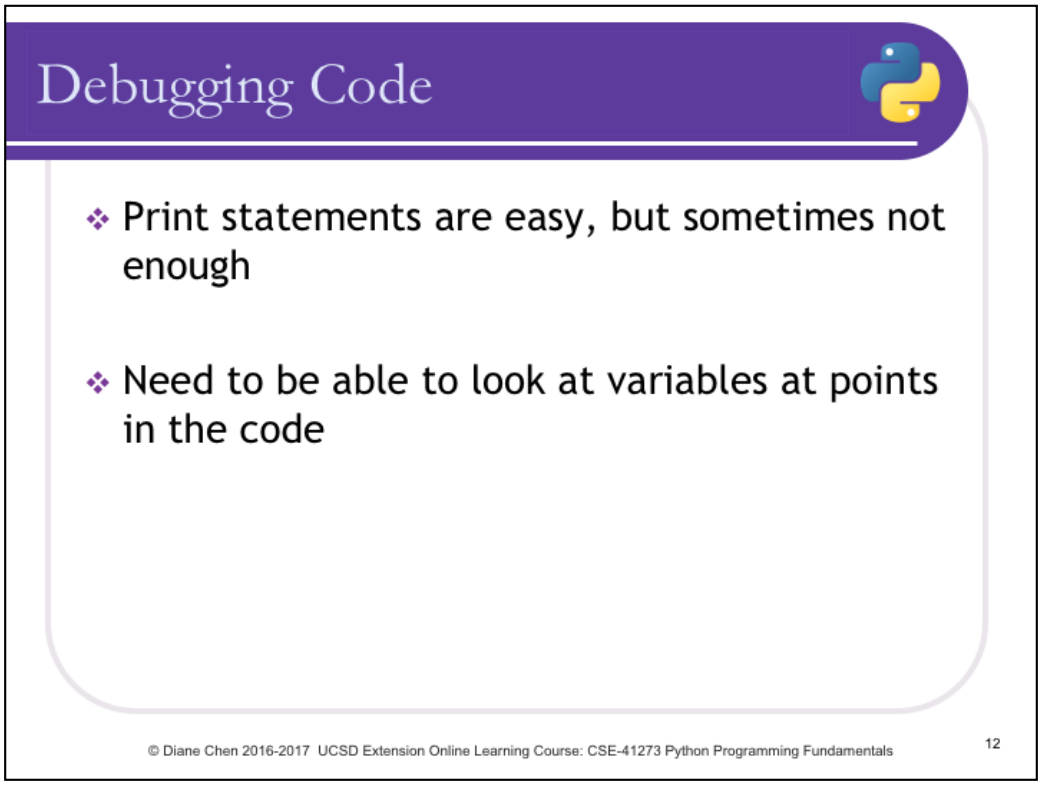
Unittest has builtin assertion methods for testing, such as assertEqual, assertNotEqual. Here I've refactored tests.py to use the assertion methods that come with unittest. Note the nice, boring output from running the tests. We never want excitement in our testing!



Here are some of the more commonly used assertion methods from unittest. You can Check the unittest documentation for more detailed information and a complete list of assertions.

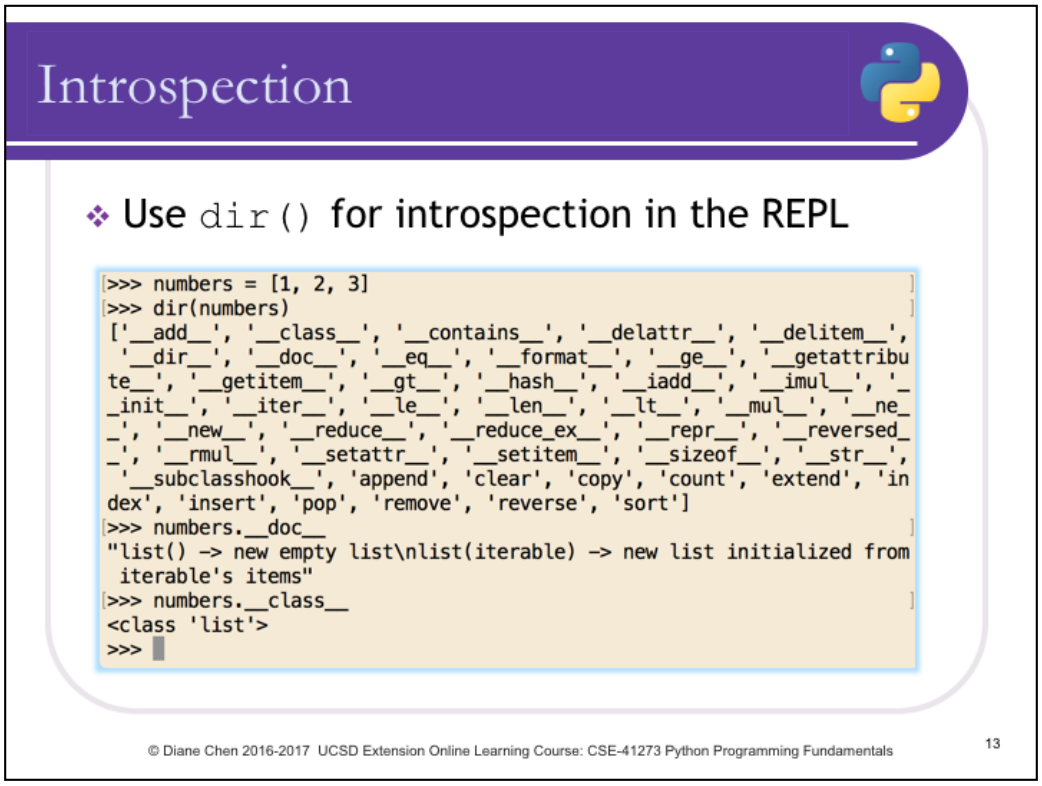
Besides the assert equal and assert not equal, I probably use the first 4 in the list the most. However, it really depends upon the type of data and functionality that is being tested.

So what do we do if we want to make sure that an exception is raised under certain conditions? We can't just test causing the error condition, because unittest will think that we had an error in our test, so it won't pass – it prints out a capital E instead of a dot or capital F when an unexpected error occurs. That won't do! One of the assertion methods of unittest is assertRaises, which is implemented as a context manager. We will talk about context managers more in a future lesson. You can think of the context manager in this case as sort of a "mini-sandbox" that we can use to encapsulate and test the exception. Here I've added a couple more tests. One tests that "addition" of strings works. The other tests that attempting to subtract two strings results in a TypeError. If a TypeError did not occur, the test would have failed.



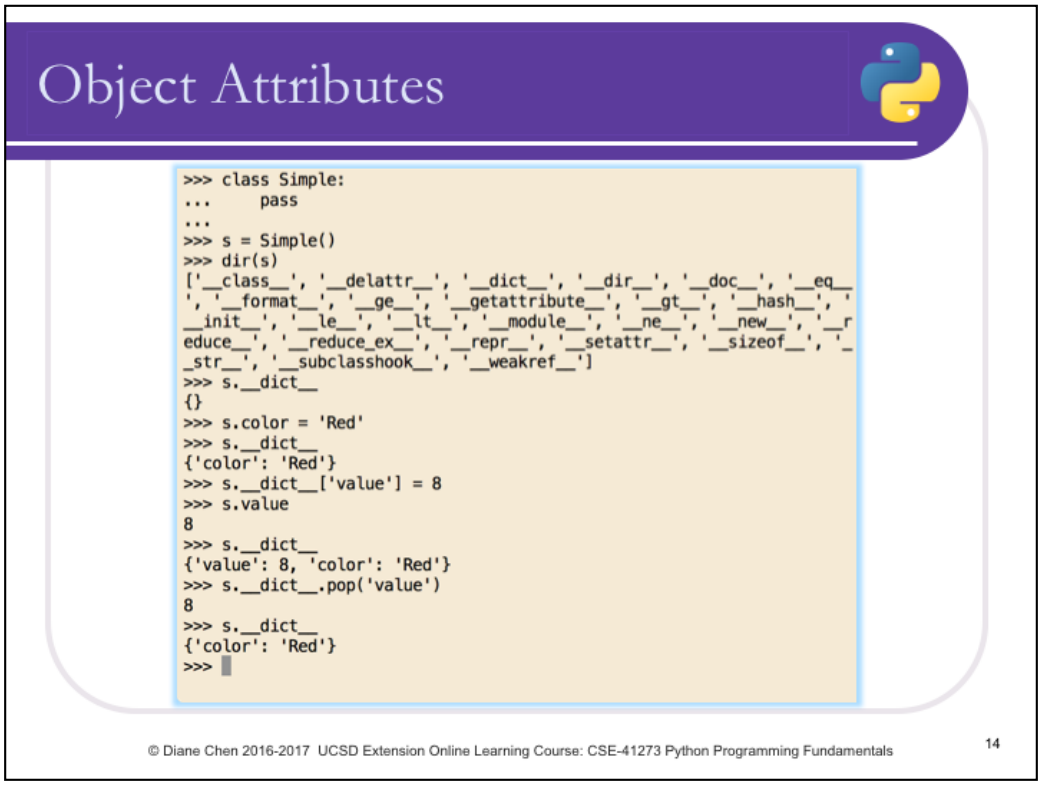
Let’s talk about debugging code. For simple programs, when we have an error, we can often get information while running the code (or parts of it) in the REPL. It’s also pretty easy to add print statements to see what’s happening in the code. But when things get more complicated, you don’t always know what to print out, or quite where to put the print statements. Sometimes there are lots of variables to look at and printing would be overload. We can use the Python debugger, pdb, to investigate errors in the code.

We'll go over a simple example of using pdb, but first let's talk a little bit about getting object information from the REPL.



First let's see some ways of using introspection on variables and things. Python has a "dir()" function that can be used for introspection at the REPL. Here we see the results of calling "dir" on a list. These are all the methods for the object. There are lots of dunder methods and attributes, aren't there? Many of them should be familiar to you. It has a dunder doc attribute for its base list function. The dunder class attribute is what gets printed when you use the function type() on an object.

Object Attributes



```
>>> class Simple:
...     pass
...
>>> s = Simple()
>>> dir(s)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__
', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '
__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__r
educe__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '_
_str__', '__subclasshook__', '__weakref__']
>>> s.__dict__
{}
>>> s.color = 'Red'
>>> s.__dict__
{'color': 'Red'}
>>> s.__dict__['value'] = 8
>>> s.value
8
>>> s.__dict__
{'value': 8, 'color': 'Red'}
>>> s.__dict__.pop('value')
8
>>> s.__dict__
{'color': 'Red'}
>>>
```


Let's look at object attributes in the REPL. We'll make a super simple class that does nothing. We instantiate a variable "s" of our Simple class. The dir() function on "s" shows lots of stuff! These come automatically when you create a class, though they aren't all actually implemented. One of the interesting attributes is dunder dict. It contains a dictionary of object attributes, which of course, is initially empty since we haven't defined any attributes. We can add an attribute 'color' to our object, and it is added to the object's dunder dict dictionary. We can treat it just like a dictionary, even adding another element to it, which adds a new attribute to the object in dunder dict. Then we can pop an attribute and it is removed from the dictionary. So this gives you an idea of the structure of an object.