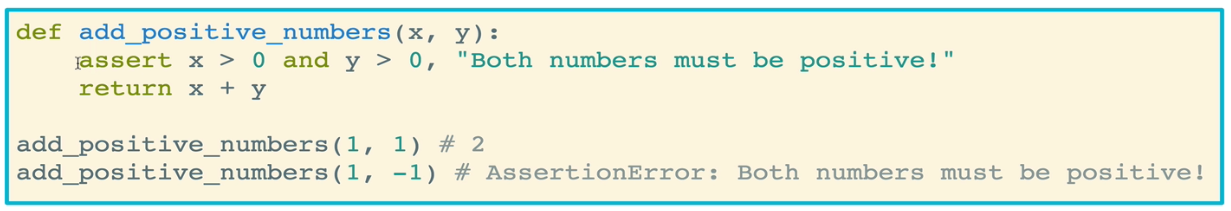
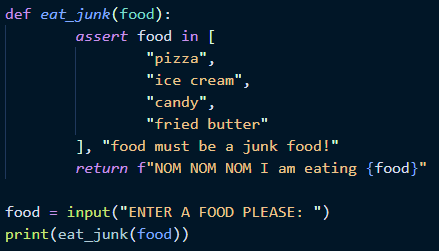
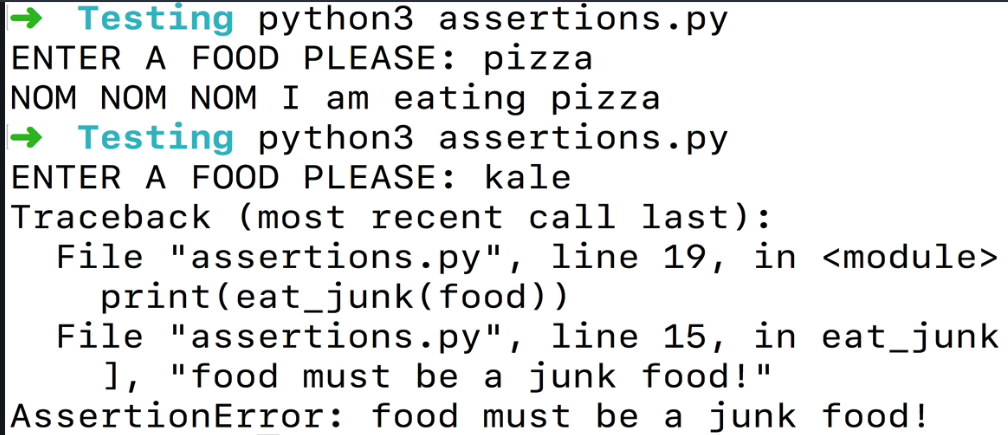
* Testing is like eating vegetables – everyone tends to agree that it’s a good idea and should be done, but many people do it only begrudgingly, or don’t do it at all
  + TDD – test-driven development, a keyword that really triggers some people negatively
  + Overall, testing can be enjoyable or unenjoyable, but it is never a bad idea
* Why should we test?
  + Testing allows you to reduce bugs in existing code
    - This class should behave this way when it’s run
    - They are particularly useful for when you have very long code where it can be challenging to find where bugs are
  + Testing also ensures that bugs that are fixed stay fixed
    - A team of developers working on one project will work on small pockets of code that they later merge into the code base. Perhaps someone changes a line of code from someone else’s piece, which then breaks the full code. Testing allows you to find out where things went wrong
  + Testing ensures that new features don’t break old ones
  + Testing ensures that cleaning up code does not introduce new bugs
  + Some people think it makes development more fun
    - This is debatable, but it definitely makes development more satisfying when your tests pass
    - On the other hand, writing tests can slow you down A LOT
* Test Driven Development
  + Testing is the idea of writing code to test other code
    - Can be written at any point, from a single line or code block to the entire application
  + **Test driven development** is different in that the idea is to write your tests first, then you write the code needed to pass the tests. After passing the tests, your code (“feature”) is considered complete
    - It is a fairly radical way of changing your coding approach, and some people very strongly abide by it
    - **Red, Green, Refactor** is the mantra of TDD. Refers to the colors of your test failing and passing, a workflow if you will
      * Red – write a test that fails
      * Green – write the minimal amount of code necessary to make the test pass
        + This allows you have the most parity as possible between the test and code
      * Refactor – clean up the code, while ensuring that the tests still pass
* **Assertions** allow you to assert that certain expressions must be true – if they are not, an assert error will be thrown
  + Uses the *assert* keyword, which accepts an expression
  + The expression returns *None* if the expression is truthy, and raises an *AssertionError* I the expression of falsy
  + Asserts also accept an optional error message as a second argument
  + Assert cannot be used to test all code because you can ignore all asserts in your Python file using one line of code that causes your code to run in optimized mode. There are newer and better options now, including *unittest*
  + Assert is NOT a function – it is a statement
  + Example of assert: This asserts that the two numbers passed into a function are both positive numbers prior to executing the rest of the function. If the assert statement is true, the custom assertion error is raised!
    - For the below example, it would be similar to writing an if statement that checks that the numbers are positive

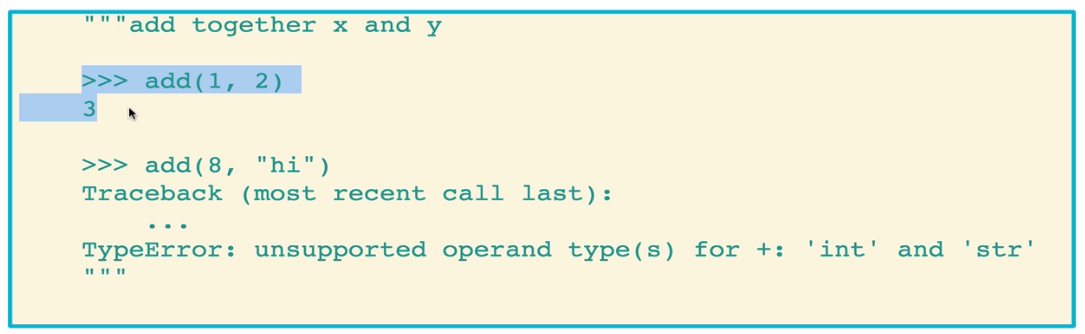


* + Another example: asserting that food passed into the function is junk food by testing the food against a list of junk foods

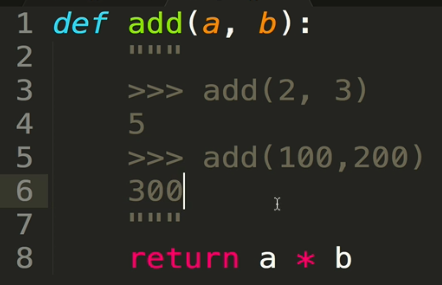


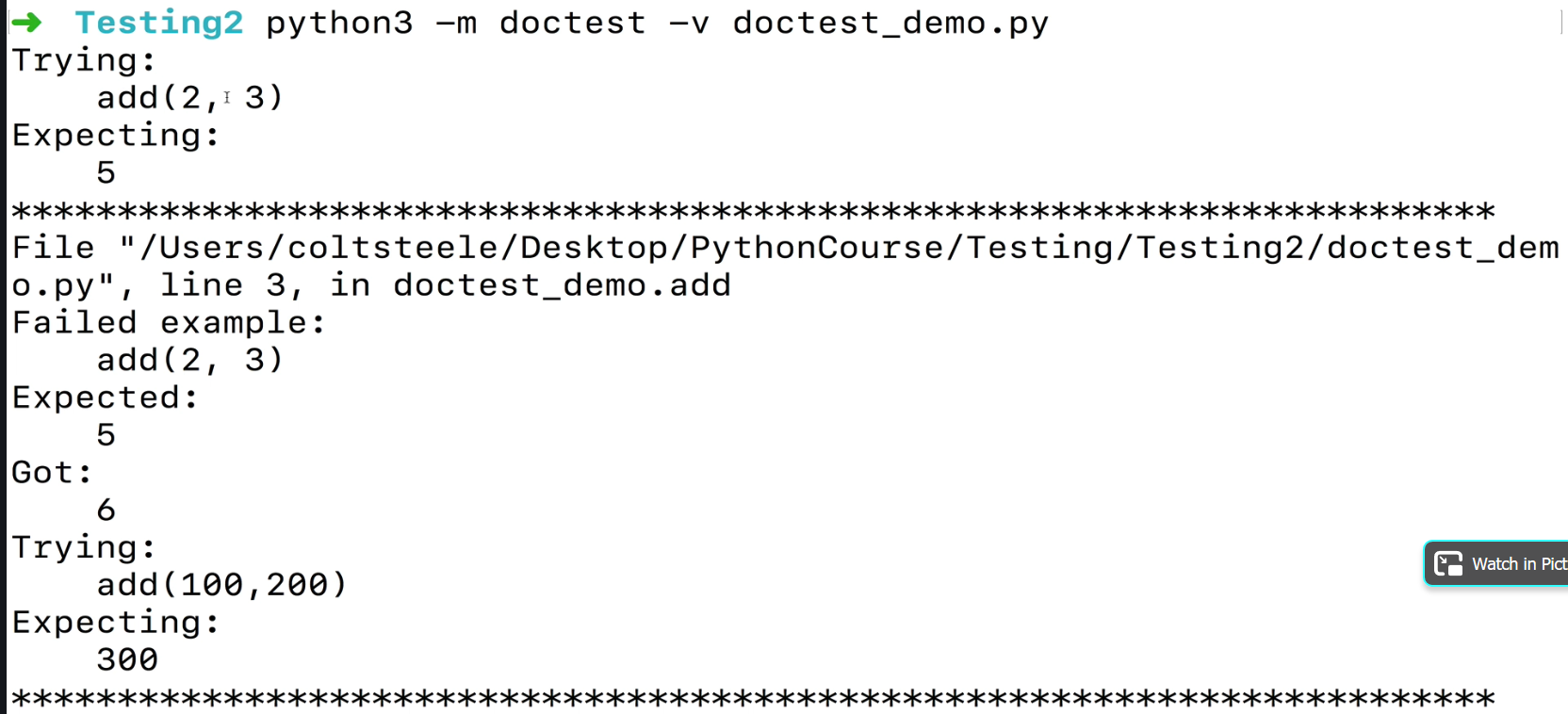


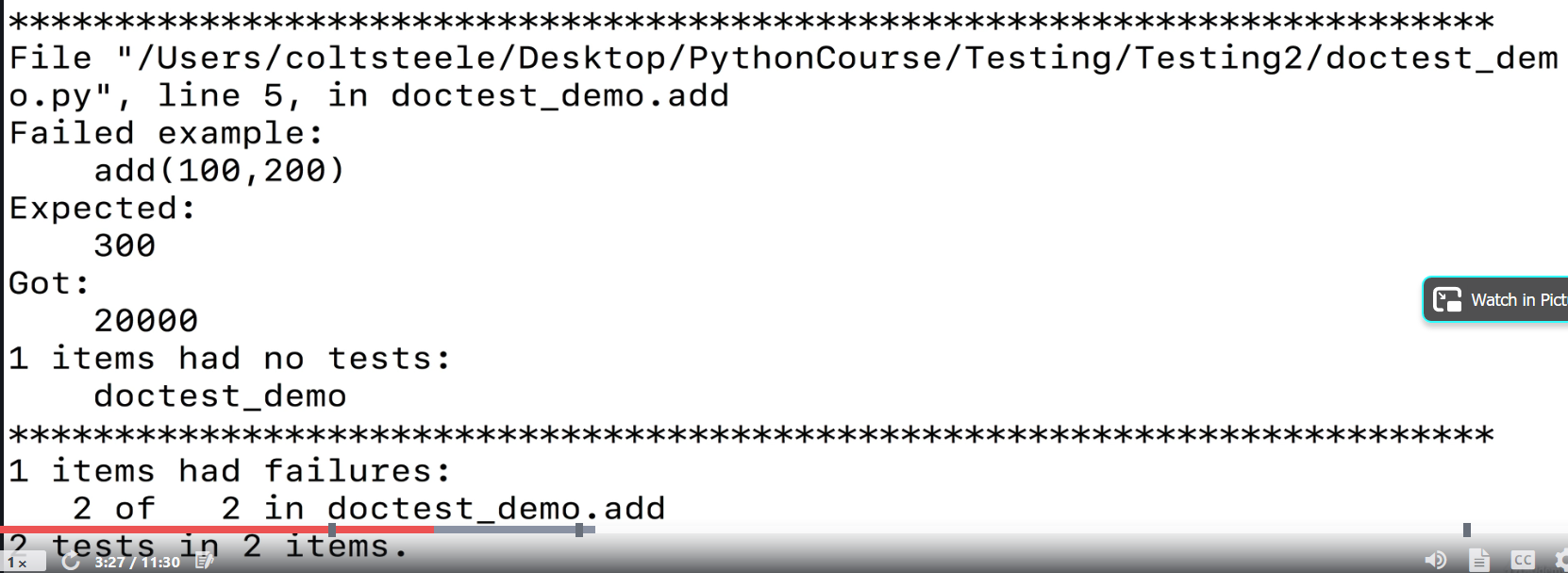
* Assertion warning: if a Python file is run with the –O flag (capital letter O, “oh”, assertions will not be evaluated. The assertions will be ignored!
  + –O stands for optimized or optimization
  + Net: asserts can be useful, but they are not guaranteed to run
* **Doctests** are a means for writing test code, any Python has a way to automatically parse and run that code for us
  + Allows us to write test for functions inside of the docstring
  + The catch is that you have to write the code so that it looks like it’s inside a REPL
  + Example:



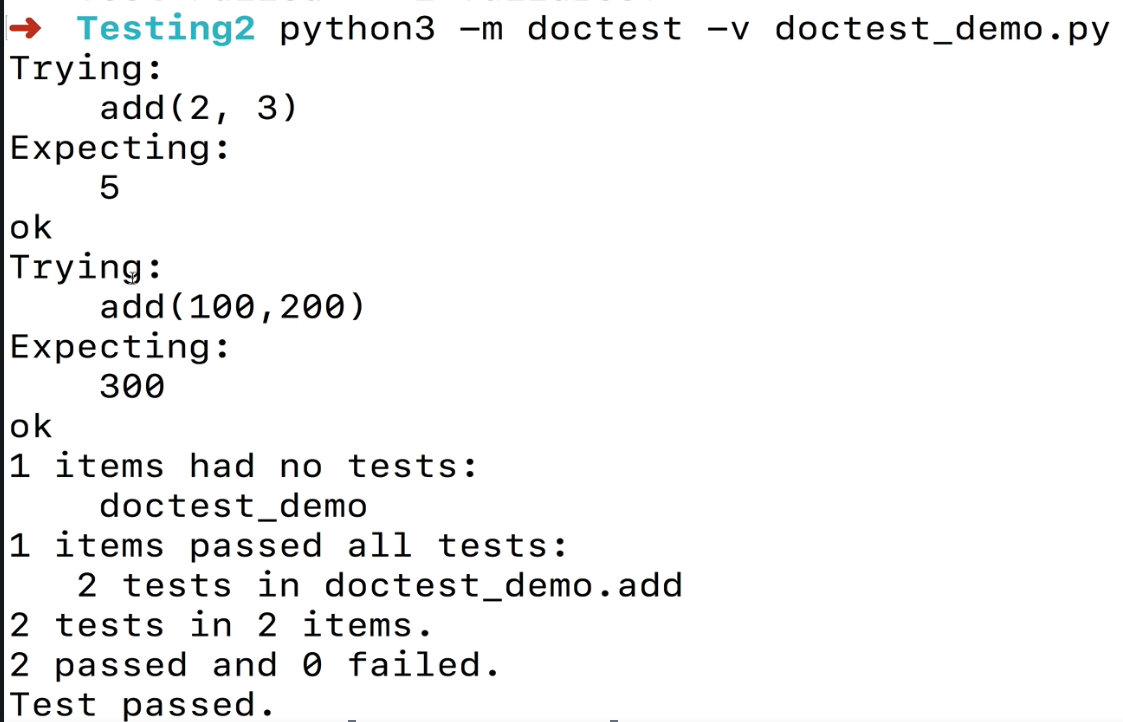
* + Another example: notice how we are actually asking the function to multiply the two numbers, but the doctest we wrote indicates that the numbers should be added. So the test will fail!



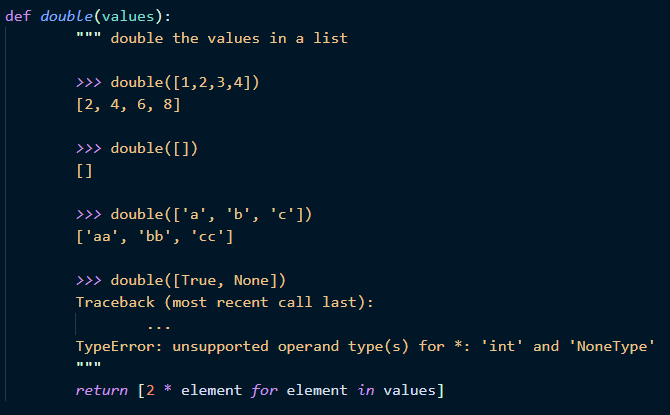




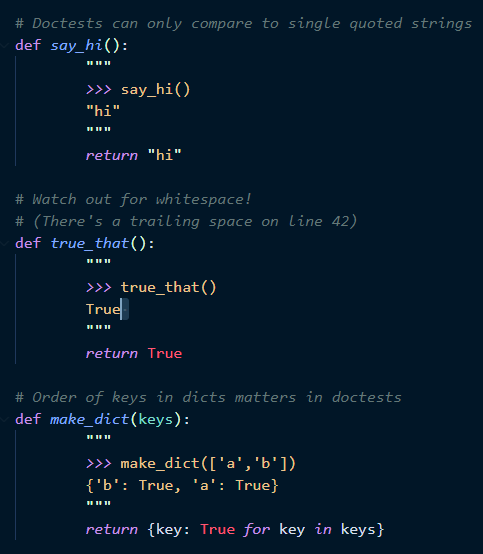
* + - After fixing the return to give a + b:



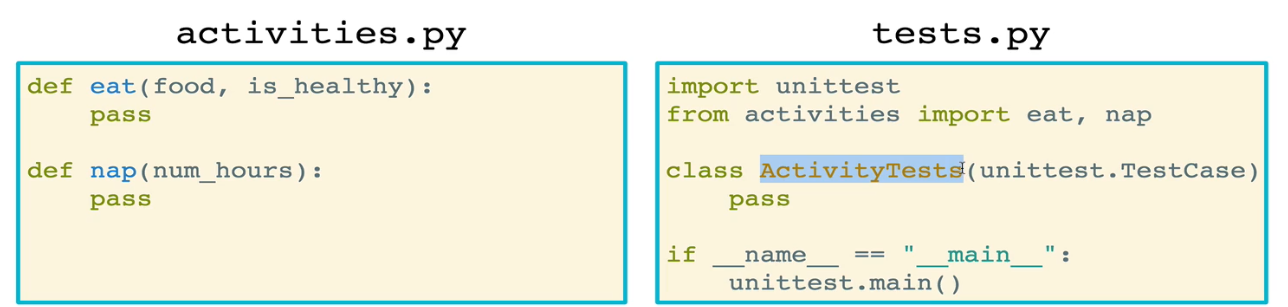
* Doing doctest the TDD way (test-driven development)
  + Remember that with TDD, you write your tests first, then construct your function or code so that it passes your tests
  + Note that writing tests that you are trying to fail (ensuring you fail when you are supposed to) requires that you EXACTLY mimic the output of the error that Python would throw



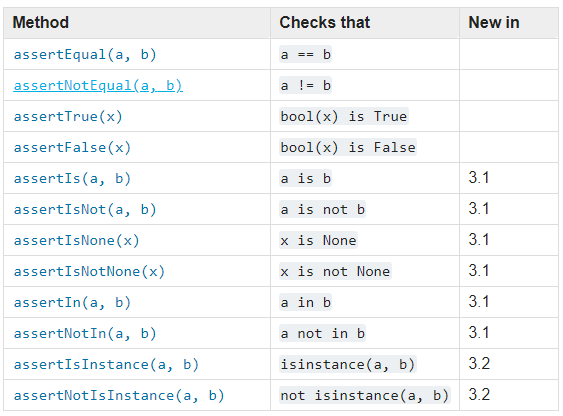
* + It can also be finicky, you need to get the syntax exactly right or it won’t work the way you need it to
    - For example, doctest will always expect strings to be in single quotes and never double quotes. If you try to put a double quoted string as an expected doctest return, the test will fail
    - Also, white space will kill you. Be aware of white space!
    - Order of keys for dictionaries matters for doctests



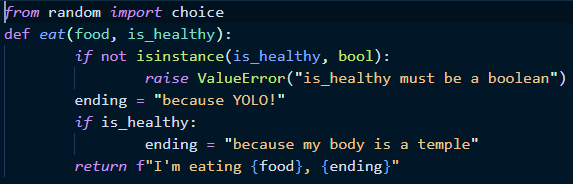
* Other drawbacks of doctests:
  + Syntax is strange
  + It clutters up your function code, making a simple function start to look very complex
  + Doctests lack several important features that other testing tools have
  + Tests can be finicky or brittle
  + They ain’t the instructor’s first choice for writing tests
* **Unittest** is the most popular testing tool and is arguably the easiest to use. What is it?
  + This is the idea of testing small individual pieces (units) of your application in isolation
  + Good candidates for unit testing include individual classes, modules, or functions
  + Bad candidates for unit testing include entire applications, and dependencies across several classes or modules
  + Unittest documentation: <https://docs.python.org/3/library/unittest.html>
* Using unittest
  + Unittest is a built-in module in Python
    - You can write unit tests encapsulated as classes that inherit from unittest.TestCase class
      * Remember inheritance? This is from the classes section, where child classes inherit the properties of parent classes
    - This inheritance give you access to many assertion helpers that let you test the behavior of your functions
    - You can run tests by calling unittest.main() at the end of your test file
  + Example of basic unittest setup
    - You can start by creating a file with your functions that you want to test (activities.py in the example below)
    - Then you create another file in which you import the unittest module as well as the functions from the first file that you want to test (tests.py in the example below). Within this second file, you set up a new class that can be called anything you want, but it must inherit the unnitest.TestCase class
      * Note that at the bottom of tests.py, the unittest.main() command runs your test



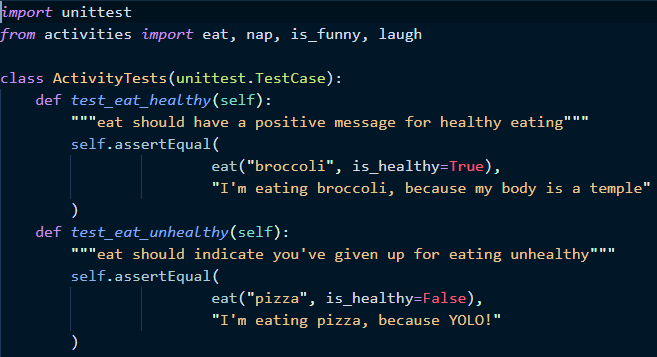
* + Assert methods



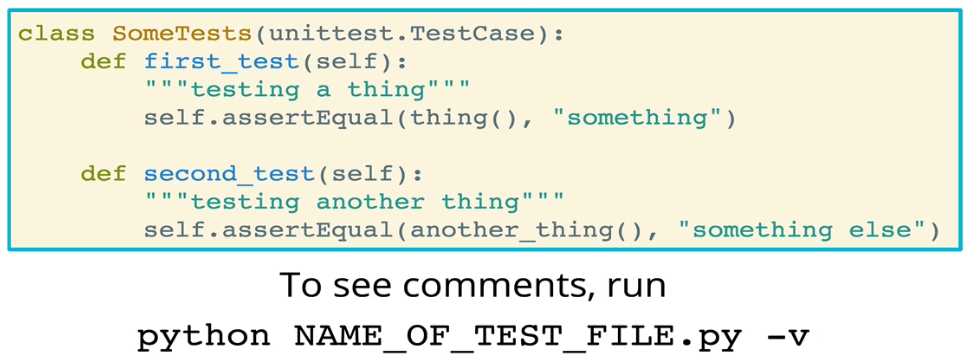
* + Next, you write your tests. What you are doing is that you are testing your functions for when you pass in certain values and checking whether you get the response that you are testing for. You can test your functions for multiple different values that you pass in, and each one of these tests can have a different name!
    - If you are utilizing TDD then you will fail all of your tests initially (be completely in the red), which is expected
    - Then you go to your functions and build them so that they pass your tests
    - Function to test



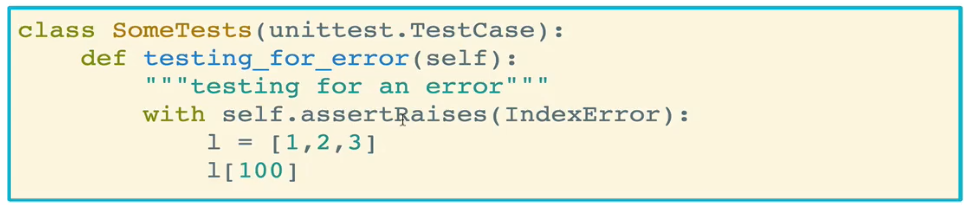
* + - Unittest (test) file. The example below utilize the assertEqual methods, which tests that the first argument and second argument resolve to the same thing



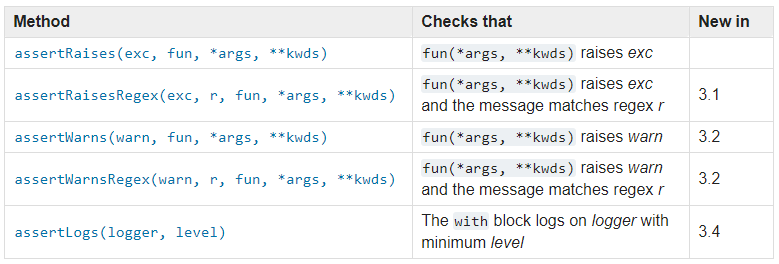
* + Tests are executed by simply running your test file
  + Tests can also be commented, and you’ll see these comments by running the verbose version of the file with –v
    - Running the verbose test without any comments will just report which tests are passing (OK) and which are failing



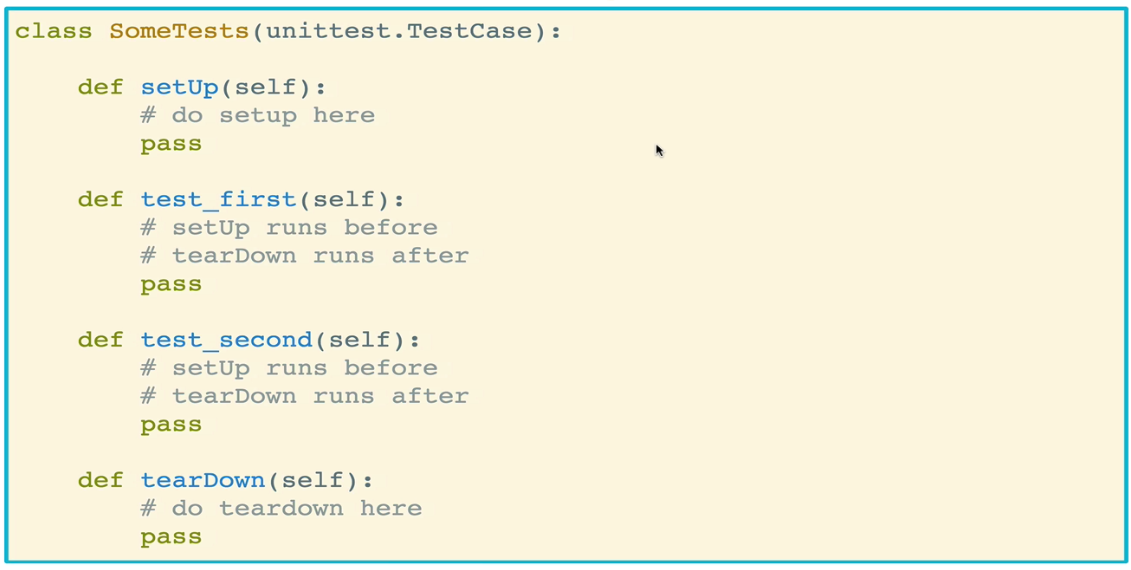
* Unittest can also be used **for testing for errors**, particularly be using methods such as self.assertRaises():
  + You can write your code to ensure that you receive an error. This is helpful when, for example, you want to enforce the type of value that a function receives
  + Example: The test below is checking whether an IndexError is raised when you attempt to access an index that is outside the range of an iterable. In order to pass this test, the condition or function must be written to make use the IndexError is actually raised



* + Table of tests for errors
    - exc = exception
    - fun = function



* Review of the basic recipe for test files
  + Define the class which must inherit unittest.TestCase
  + Define the test functions within the class. Each test case must have at least one assertion statement inside of it
  + Call unittest.main() after the your class definition
* Before and After Hooks in Unittest
  + **setUp** and **tearDown** are useful for running code before and after each of your tests. It would be a pain to have to write that code in every single test function. setUp and tearDown offer an alternative
  + Common use cases include:
    - Adding or removing data from a test database
    - Creating instances of a class
  + Useful for testing larger applications, where you may want to ensure a similar application state before running tests
  + setUp runs before each test method
  + tearDown runs after each test method; this is typically not common unless you are using a database



* + One useful application of setUp is to reset a common instantiation of a class when running tests on the same instance of that class

