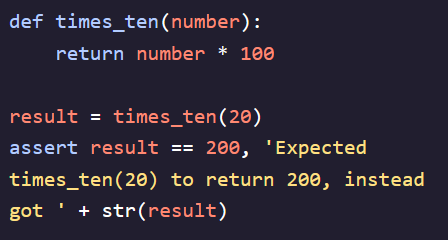
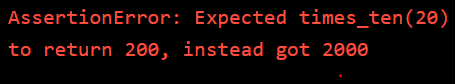
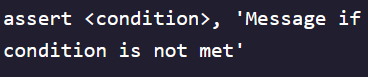
**Testing:**

- Crucial to creating quality software  
- Goal of testing isn’t just to find bugs but to find them quickly  
- Tests can be divided into two categories:  
- ***Manual Testing -***  A physical person interacts with software in the same way a user would  
­ ***Automated Testing*** – Tests that are performed with code, faster and less prone to human error

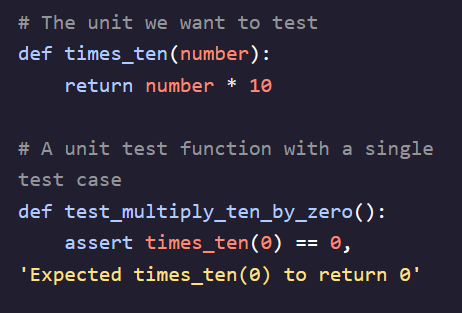
**Assert Statement:**

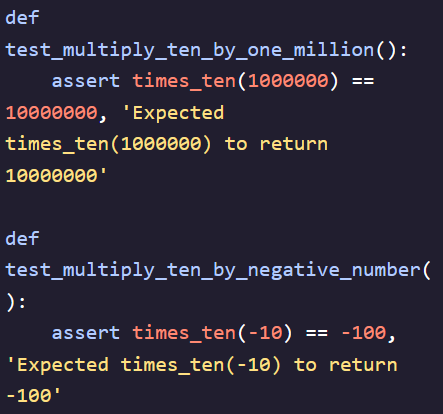
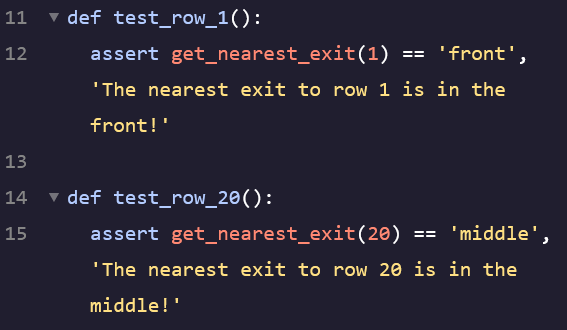
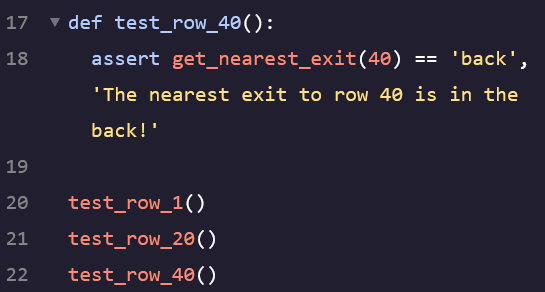
- Performing manual checks over large amounts of code is tedious and we should write automated testing to do it for us  
- An ***assert***  statement can be used to test that a condition is met. If a condition evaluates to *False* an ***AssertionError*** is raised with an optional error message  






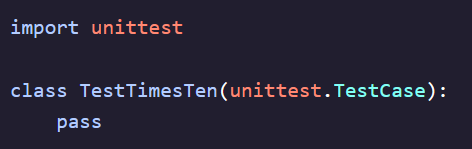
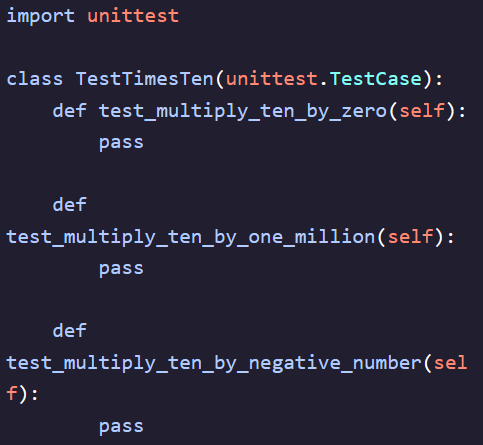
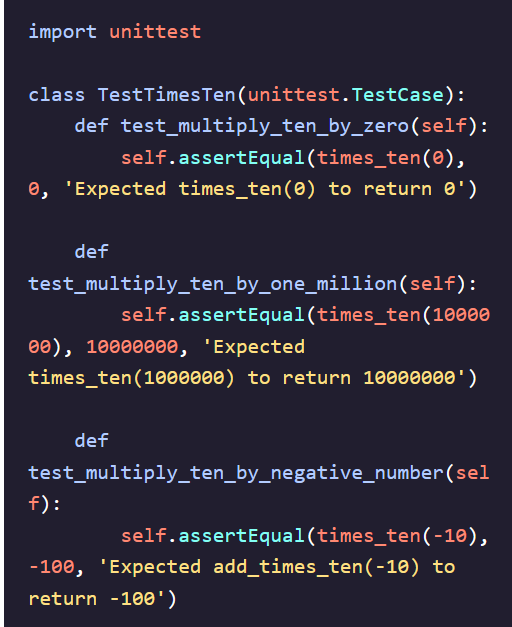
**Unit Testing:**

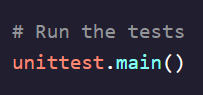
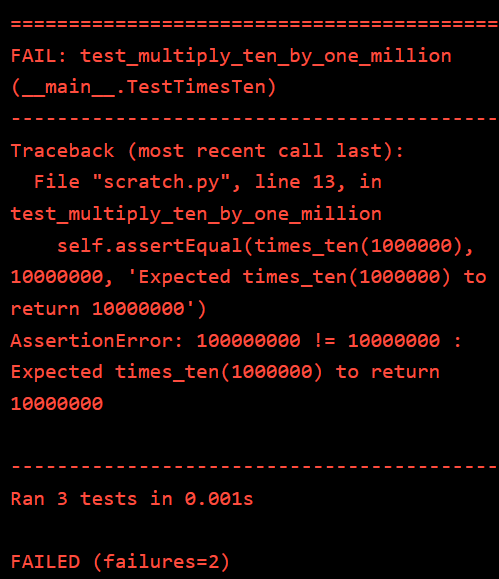
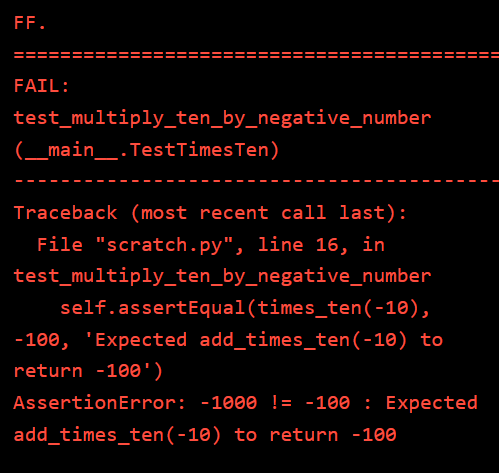
- Testing begins at the smallest unit of a program (function, loop, or variable)  
- A unit test validates a single behavior and will make sure all of the units of a program are functioning properly   
- ***Test Case*** – Validates that a specific set of inputs produces the expected output for the unit we are testing  


- Best practice is to create test cases for specific edge case inputs as well as reasonable ones (large number, negative number, and zero)  
  

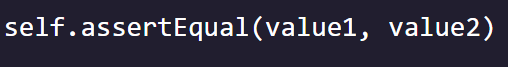
**Unittest Framework:**

- Testing one by one is burdensome because:   
1). We have to call each function specifically when a new test is created  
2). No way of grouping tests together, necessary when number of tests increases  
3). If one test fails, *AssertionError* prevents any remaining tests from running  
  
- Python provides the ***unittest*** framework to solve these problem and provide other tools for writing unit tests  
- Need to import the module into code when using ***import unittest***- Provides us with a ***test runner*** – A component that collects and executes tests and then provides results to the user. Also provides other tools for test grouping, setup, teardown, and skipping  
**Creating a Unittest:**

- First, we need to create a class which inherits from *unittest.TestCase* to server as main storage of all unit testing functions  
- Then, we need to create functions within the class. These are required to begin with the word *test*- Lastly, we need to use *assertEqual* statements to delineate tests  


- Run code by calling *unittest.main()* and it will detect any tests in existing module, run them, and provide results  
 

**Assert Methods - Equality and Membership:**

- ***assertEqual()*** – Takes two values as arguments and checks if they are equal, if they are not the test fails  
  
- ***assertIn()*** – Takes two arguments and checks that the first argument is found in the second argument, which should be a container. If it’s not in the container, the test fails  
A white text on a blue background

Description automatically generated  
- ***assertTrue()*** – Takes a single argument and checks to see that the argument evaluates to ***True***. If it doesn’t, then the test fails  
A white text on a blue background

Description automatically generated

A screenshot of a computer program

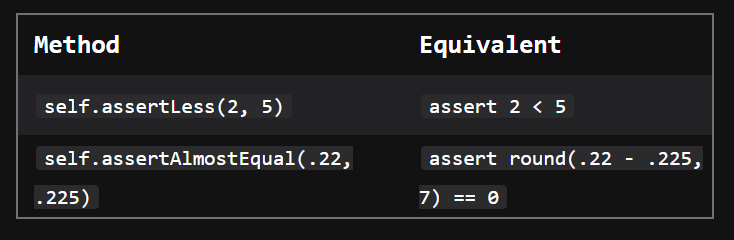
Description automatically generated

**Assert Methods – Quantitative Methods:**

- ***assertLess()*** – Takes two arguments and checks that the first argument is less than the second one. If not, then the test fails  
A close up of a logo

Description automatically generated  
- ***assertAlmostEquals()*** – Takes two arguments and checks that their difference, when rounded to 7 decimal places, is 0. If the values are not close enough to zero, then the test will fail  
A blue and white text

Description automatically generated



**Assert Methods – Exceptions and Warning Methods:**

- ***assertRaises()* ­**– Takes an exception type as its first argument (KeyError, TypeError), a function reference as its second argument, and an arbitrary number of arguments as the rest

- Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn’t warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

A black background with white text

Description automatically generated  
A screen shot of a computer code

Description automatically generated

- ***assertWarns()*** – Takes a warning type as its first argument, a function reference as its second argument, and an arbitrary number of arguments as the rest

A close up of white text

Description automatically generated  
A screen shot of a computer code

Description automatically generated

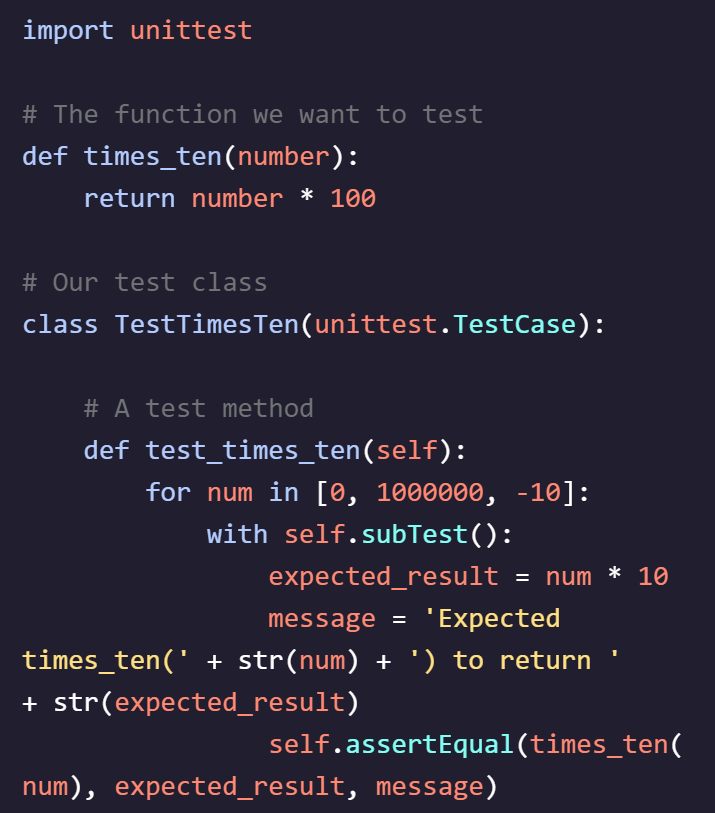
**Parameterizing Tests:**

- To decrease repetition, Python provides us a specific toolset for tests with only minor differences, this is known as ***parameterization***

- By parametrizing tests, we can leverage the functionality of a single test to get a large amount of coverage of different inputs

- The ***unittest***framework provides us with the ***subTest*** context manager (A protocol supported by objects that enables them to be used within a with statement. It's particularly useful when you need to manage resources, such as opening and closing files, acquiring and releasing locks, or handling exceptions) to accomplish this

- By using ***subTest*** each iteration of the loop is treated as an individual test. Python will run the code inside of the context manager on each iteration and return the failure as a separate test case failure

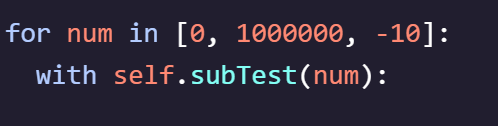
 A computer screen with red text

Description automatically generated

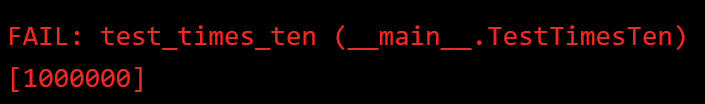
A screenshot of a computer program

Description automatically generated

Can make code clearer using the below syntax



Returns an easier to understand error message



**Test Fixtures:**

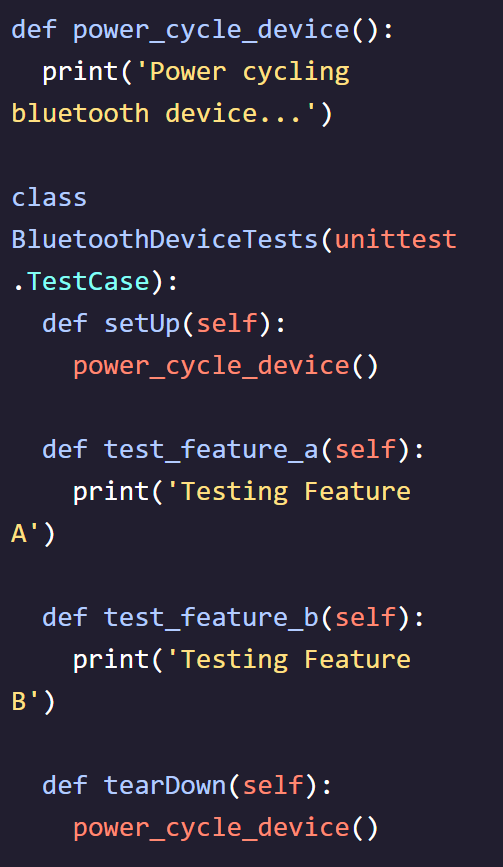
- Tests need to occur in a known state, if the conditions in which a test runs are not controlled then the results could contain false negatives or false positives

- A ***test fixture*** is a mechanism for ensuring the proper test setup (putting tests into a known state) and test teardown (restoring the state prior to the test running).

- ***Test fixtures*** guarantee that our tests are running in predictable conditions

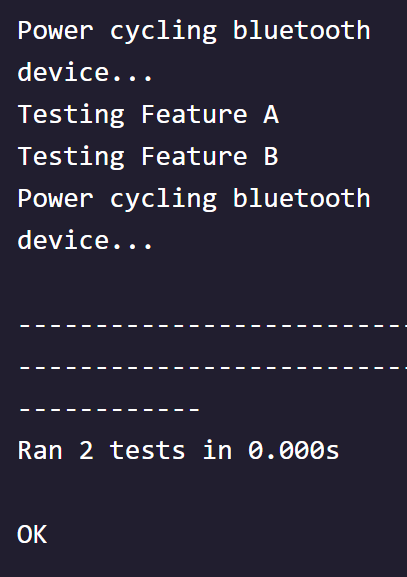
- In this scenario we want to ensure the device is not in a failed state already when the test begins to run so we power cycle it each time  
  
- The ***unittest*** framework automatically identifies setup and teardown methods based on their names

- A method named ***setUp*** runs before each test case in the class and a method named ***tearDown*** gets called after each test case

 A screenshot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated 

- In this scenario we rely on working Bluetooth and there is nothing in the tests that would cause the Bluetooth to stop working. Setup and Teardown only need to happen once

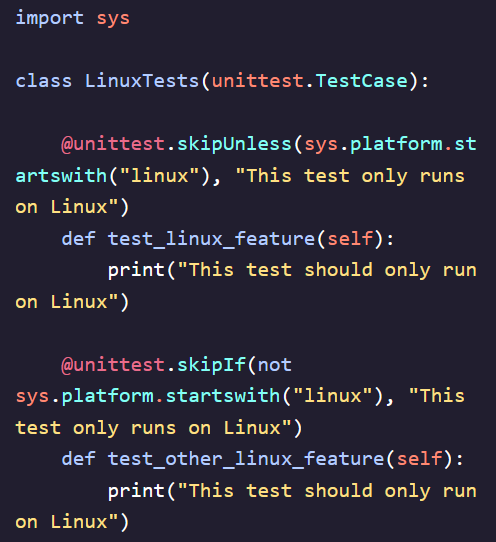
- We replaced our ***setUp*** method with the ***setUpClass*** method and added the ***@classmethod*** decorator. We changed the argument from self to cls because this is a class method. Similarly, we replaced the ***tearDown*** method with the ***tearDownClass*** class method.

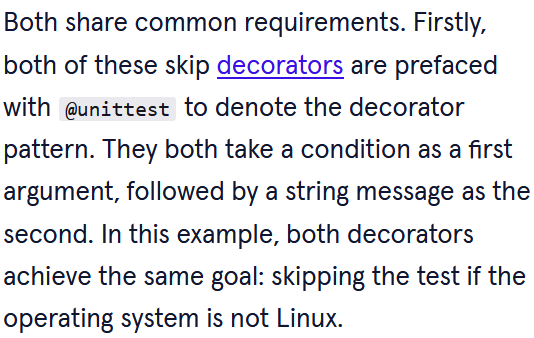
**Skipping Tests:**

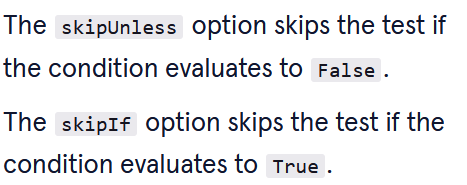
- Sometimes we want our tests to only run in a particular context (only want them to run on Windows and not Linux or macOS

- ***unittest*** provides two different ways to skip tests:

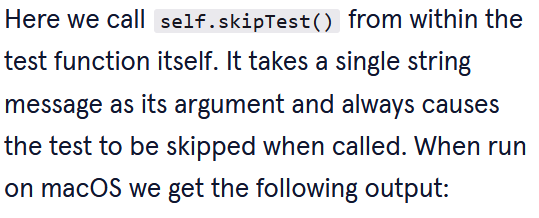
1. ***@unittest.skipUnless() / @unittest.skipIf()*** decorators

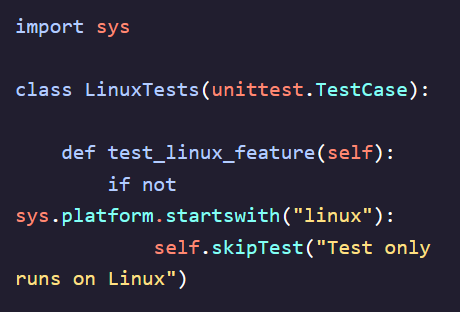






2. ***skipTest()***





**Expected Failures:**

- Sometimes we have a test that we know will fail, this could be due to a known bug or its designed to fail on purpose

- Rather than skipping this test, ***unittest*** provides a way to mark the test as an expected failure.

- Expected failures are counted as passed in our test results. If the test passes when we expected it to fail then it is marked as failed in test results



- Use the ***@unittest.expectedFailure*** decorator

