Unit Testing

`unittest` Module

In Python, the `unittest` module from the standard library provides a framework for unit testing.
It can be imported with `import unittest`.

```
import unittest
```

Skip with `unittest`

In Python's unittest framework, individual test methods can be skipped by using the `skip`, `skipIf`, or `skipUnless` decorators, as well as the TestCase `.skipTest` method.
The `skipUnless` option skips the test if the condition evaluates to `False`. The `skipIf` option skips the test if the condition evaluates to `True`.
In the code example, both `skipUnless()` and `skipIf()` skip the test if the operating system is not Windows.

```
class Tests(unittest.TestCase):


@unittest.skipUnless(sys.platform.starts
with("win"), "requires Windows")
    def test_linux_feature(self):
        print("This test should only run
on windows")

    @unittest.skipIf(not
sys.platform.startswith("win"),
"requires Windows")
    def test_other_linux_feature(self):
        print("This test should only run
on windows")
```

`expectedFailure` with `unittest`

In Python's unittest framework, individual tests can be marked as an expected failure using the `expectedFailure` decorator. With this decorator, the test will be considered successful in test results if it fails and will be marked as a failure if it passes. In the code example, the `.expectedFailure` decorator is used, and therefore, the `test_something_that_fails()` test will pass if `assertEqual(0,1)` fails (it indeed does fail and the test as a whole passes).

```
class Tests(unittest.TestCase):

    @unittest.expectedFailure
    def test_something_that_fails(self):
        self.assertEqual(0, 1)
```

Test Fixture

In Python, a test fixture is a mechanism for ensuring that proper test setup and teardown occur.
Test fixtures can be created by adding `setUp()` and `tearDown()` methods to the `TestCase` child class. Text fixtures created like this run before and after each individual test.
In the code example, `'This runs before each test'` will be printed before each test in the class, and `'This runs after each test'` will be printed after each test.

```
class Tests(unittest.TestCase):

    def setUp(self):
        print('This runs before each
test')

    def test_equality(self):
        self.assertEqual(1, 1)

    def test_list(self):
        self.assertIn(1, [1, 2, 3])

    def tearDown(self):
        print('This runs after each
test')
```

SyntaxError

A `SyntaxError` is reported by the Python interpreter when some portion of the code is incorrect. This can include misspelled keywords, missing or too many brackets or parentheses, incorrect operators, missing or too many quotation marks, or other conditions.

```
age = 7 + 5 = 4
```

```
File "<stdin>", line 1
SyntaxError: can't assign to operator
```

`assert` Keyword

In Python, the `assert` keyword is used to test that a condition is met. If the condition evaluates to `False`, an `AssertionError` is raised with the specified error message.
The example code would produce the following output:

```
number = 5
assert number == 3, 'Number is not equal to 3!'
```

```
Traceback (most r
  File "/Users/me/
    assert number
AssertionError: Nu
```

Unit Test

In Python, a unit test validates the behavior of a single isolated component, such as a function. Often, a function is tested for its expected return value by giving it one or two extreme inputs and one normal use case.

Test Case from `unittest.TestCase`

In Python, a test case is an individual test for an expected response to a given set of inputs.
In the code example, notice how test cases are created by defining a class that inherits from `unittest.TestCase`. Then, tests are added by defining methods that start with the word test.

```
class MyTests(unittest.TestCase):

    def test_example(self):
        print('This is a test method that will pass')
```

Test Runner with `main()`

In Python, a test runner is a component that collects and executes tests and then provides results to the user.
Tests can be run by calling `unittest.main()` in a module that has a test class derived from `unittest.TestCase`.

```
class MyTests(unittest.Tes

    def test_example(self)
        print('This is a t

unittest.main()
```

The example code would output:

```
.
----------------------------
Ran 1 test in 0.000s

OK
```

`assertEqual()` from `unittest`

In Python's unittest framework, the `assertEqual` TestCase method is used to test that two values are equal to each other.
In the code example, `.assertEqual()` is checking if `output` (which is `add_five(1)`) is equal to `6`.

```
class Tests(unittest.TestCase):

    def test_add_five(self):
        output = add_five(1)
        self.assertEqual(output, 6)
```

`assertIn()` from `unittest`

In Python's unittest framework, the `assertIn` TestCase method is used to test that a specific item is present in a container.
In the example code, `.assertIn()` checks if `'red'` is present in `colors`. In this case, the test will fail since `'red'` is not included in the list returned by `get_rainbow_colors()`.

```
def get_rainbow_colors():
  return ['orange', 'yellow', 'green',
'blue', 'indigo', 'violet']

class Tests(unittest.TestCase):

    def test_get_rainbow_colors(self):
        colors = get_rainbow_colors()
        self.assertIn('red', colors)
```

`assertRaises()` from `unittest`

In Python's unittest framework, the `assertRaises` TestCase method is used to test that a specific exception is raised.
The `assertRaises` method takes an exception type as its first argument, a function reference as its second, and an arbitrary number of arguments as the rest.
In the code example, the `assertRaises()` test calls the `raise_index_error()` function. If an `IndexError` exception is raised, the test passes. If any other type of exception is raised or if no exception is raised, the test will fail.

```
class Tests(unittest.TestCase):

    def test_raises_index_error(self):
        self.assertRaises(IndexError,
raise_index_error)
```

Test Fixtures with `setUpClass()` and `tearDownClass()`

In Python's unittest framework, test fixtures can be created by adding `setUpClass()` and `tearDownClass()` methods to the `TestCase` child class. Text fixtures created like this run only once - before and after all tests in the class are executed. These methods must be decorated with the `classmethod` decorator.
In the code example, `'This runs once before all tests'` will be printed once before all the tests in the class, and `'This runs once after all tests'` will be printed after all the tests.

```
class Tests(unittest.TestCase):

    @classmethod
    def setUpClass(self):
        print('This runs once before all
tests')

    def test_equality(self):
        self.assertEqual(1, 1)

    def test_list(self):
        self.assertIn(1, [1, 2, 3])

    @classmethod
    def tearDownClass(self):
        print('This runs once after all
tests')
```