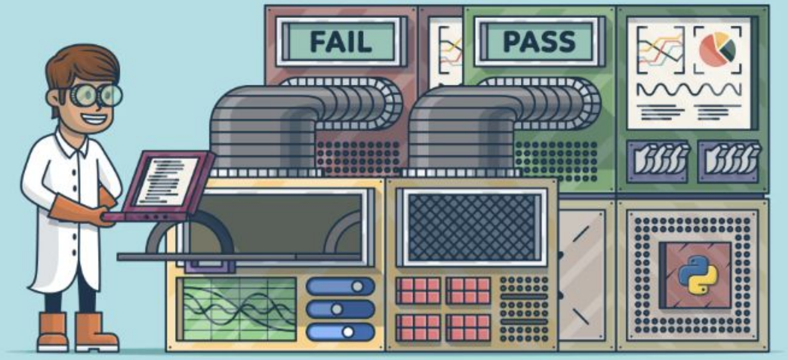
Testing in Python

Getting Started With Testing in Python

Agenda



Unit Test

Unit Tests vs Integration Tests

Test:
Test the lights on a car.

Test Step:
Turn on the lights.

Test Assertion:
Go outside of the car or ask a friend to check the lights are on.

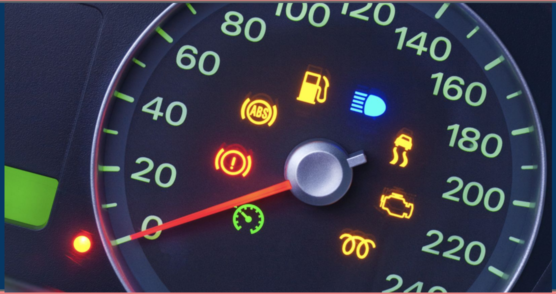
Testing multiple components is know as **Integration Test**

Unit Tests vs Integration Tests

What is wrong?

- Bulbs are broken.
- Is the battery dead.
- Alternator failure.
- Car's computing failing.

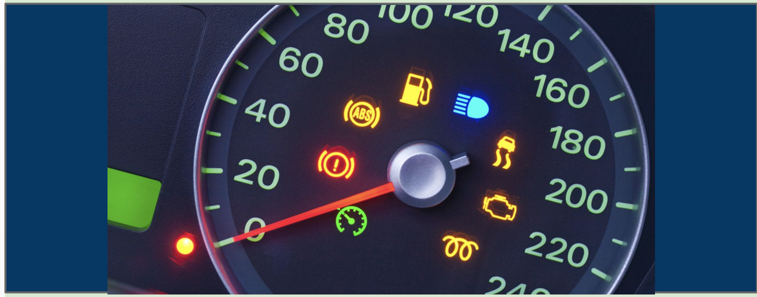
Unit Tests vs Integration Tests



Unit Tests:

- ✗ Bulbs.
- ✓ Battery.
- ✓ Alternator.
- ✓ Car's computing.

Unit Tests vs Integration Tests



Unit Tests:

- ✓ Bulbs.
- ✓ Battery.
- ✓ Alternator.
- ✓ Car's computing.

Unit Test

```python
def test_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"

def test_sum_tuple():
    assert sum((1, 2, 2)) == 6, "Should be 6"

if __name__ == "__main__":
    test_sum()
    test_sum_tuple()
    print("Everything passed")
```

```
$ python test_sum_2.py
Traceback (most recent call last):
  File "test_sum_2.py", line 9, in <module>
    test_sum_tuple()
  File "test_sum_2.py", line 5, in test_sum_tuple
    assert sum((1, 2, 2)) == 6, "Should be 6"
AssertionError: Should be 6
```

Integration Test vs Unit Test

A unit test is a smaller test, one that checks that a single component operates in the right way. A unit test helps you to isolate whats is broken in your application and fix it faster.

1. An Integration test checks that components in you application operate wich other.
2. A unit test checks a small component in your application.

Testing Your Code

Choosing a Test Runner

01 **unittest**

- Contains both a testing framework and a test runner.
- Requires
  - Put your tests into classes as methods.
  - Use a serie of special assertion methods instead of the built-in assert statement..

```python
import unittest


class TestSum(unittest.TestCase):

    def test_sum(self):
        self.assertEqual(sum([1, 2, 3]), 6, "Should be 6")

    def test_sum_tuple(self):
        self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")

if __name__ == '__main__':
    unittest.main()
```

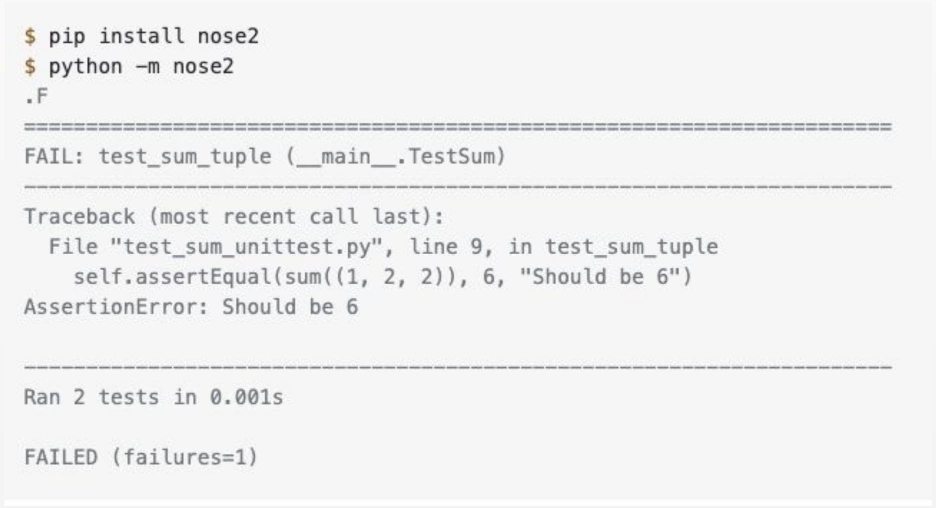
```
$ python test_sum_unittest.py
.F
======================================================================
FAIL: test_sum_tuple (__main__.TestSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum_unittest.py", line 9, in test_sum_tuple
    self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")
AssertionError: Should be 6

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Choosing a Test Runner

02

nose or nose2

- Nose is compatible with any tests written using the unittest framework.



```
$ pip install nose2
$ python -m nose2
.F
======================================================================
FAIL: test_sum_tuple (__main__.TestSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum_unittest.py", line 9, in test_sum_tuple
    self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")
AssertionError: Should be 6

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Choosing a Test Runner

03

pytest

- Support the execution of unittest test cases.
- Advantage of Pytest comes by writing pytest test cases.
- Features:
  - Support for the built-in assert statement instead of using special self.assert*().
  - Support for filtering for test cases.
  - Ability to rerun from the last failing test.
  - An ecosystem of hundreds of plugins to extend the functionality

```
def test_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"

def test_sum_tuple():
    assert sum((1, 2, 2)) == 6, "Should be 6"
```

How to Structure a Simple Test

Decision

1. What do you want to test?
2. Are you writing a unit test or an integration test?

Workflow

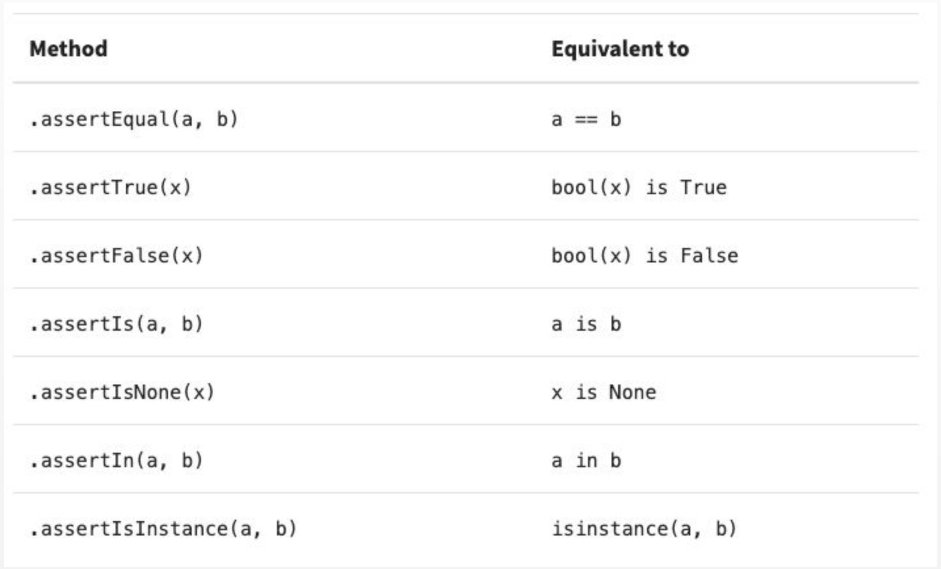
- Create your inputs.
- Execute the code, capturing the output.
- Compare the output with an expected result.

Explore the behaviors of the function

- Explore all the possible outputs of the function being tested.
- Test the errors.

How to Write Assertions

- Make sure tests are repeatable and run you test multiple times to make sure it gives the same result every time.
- Try and assert results that relate to your input data.



Method	Equivalent to
`.assertEqual(a, b)`	`a == b`
`.assertTrue(x)`	`bool(x) is True`
`.assertFalse(x)`	`bool(x) is False`
`.assertIs(a, b)`	`a is b`
`.assertIsNone(x)`	`x is None`
`.assertIn(a, b)`	`a in b`
`.assertIsInstance(a, b)`	`isinstance(a, b)`

Isolating Behaviors in Your Application

There are some simple techniques you can use to test parts of your application that have many side effects:

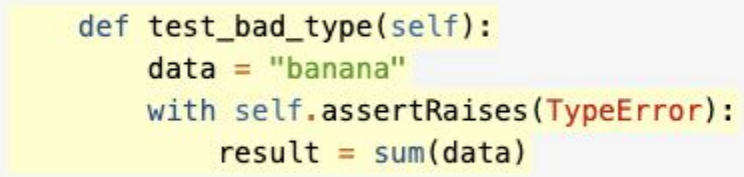
- Refactoring code to follow the Single Responsibility Principle.
- Mocking out any method or function calls to remove side effects.
- Using integration testing instead of unit testing for this piece of the application.

Handling Expected Failures

What happens when you provide in the `sum()` function a bad value, such as a single integer or a string?

In this case, you would expect `sum()` to throw an error. When it does throw an error, that would cause the test to fail.

There's a special way to handle expected errors. You can use `.assertRaises()` as a context manager, then inside the `with` block execute the test steps:



```
    def test_bad_type(self):
        data = "banana"
        with self.assertRaises(TypeError):
            result = sum(data)
```

Where to Write the Test

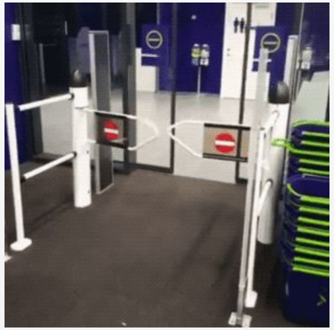
```
project/
│
├── my_app/
│   └── __init__.py
│
└── tests/
    │
    ├── unit/
    │   ├── __init__.py
    │   └── test_sum.py
    │
    └── integration/
        ├── __init__.py
        └── test_integration.py
```

Writing Integration Tests

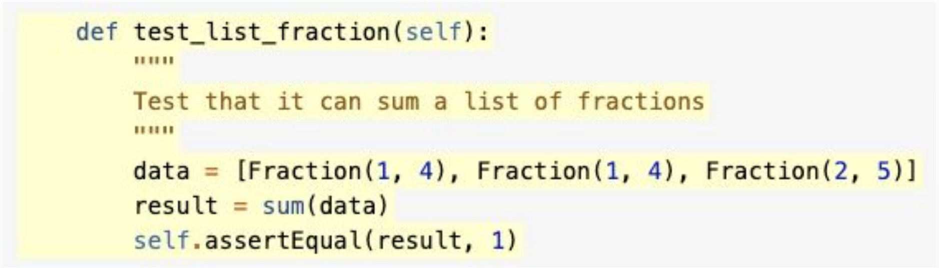
Integration testing is the testing of multiple components of the application to check that they work together. Integration testing might require acting like a consumer or user of the application by:

- Calling an HTTP REST API
- Calling a Python API
- Calling a web service

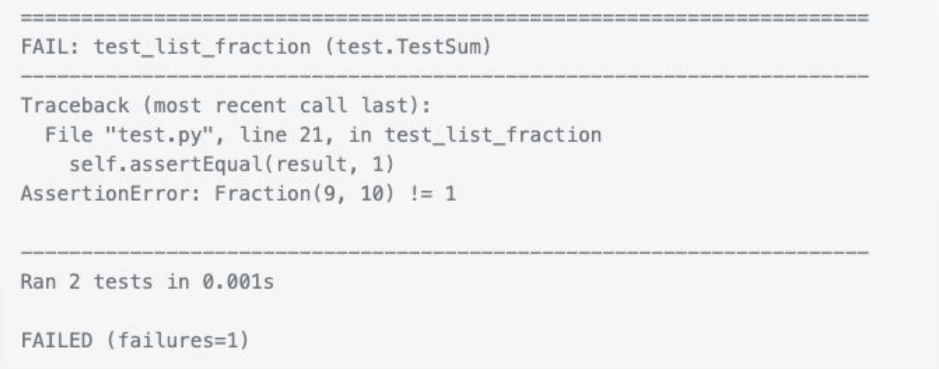
Each of these types of integration tests can be written in the same way as a unit test, following the Input, Execute, and Assert pattern. The most significant difference is that integration tests are checking more components at once and therefore will have more side effects than a unit test.



Understanding Test Output



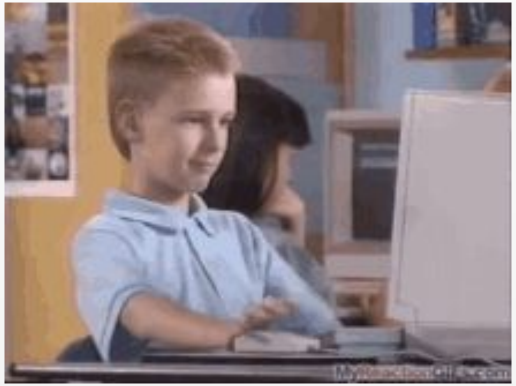
```
    def test_list_fraction(self):
        """
        Test that it can sum a list of fractions
        """
        data = [Fraction(1, 4), Fraction(1, 4), Fraction(2, 5)]
        result = sum(data)
        self.assertEqual(result, 1)
```



```
======================================================================
FAIL: test_list_fraction (test.TestSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test.py", line 21, in test_list_fraction
    self.assertEqual(result, 1)
AssertionError: Fraction(9, 10) != 1

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

1. The first line shows the execution results of all the tests, one failed (F) and one passed (.).
2. The FAIL entry shows some details about the failed test:
   a. The test method name (test_list_fraction).
   b. The test module (test) and the test case (TestSum).
   c. A traceback to the failing line.
   d. The details of the assertion with the expected result (1) and the actual result (Fraction(9, 10)).

Training Time!



Thanks!