bianca.phelan@tudublin.ie

O Object
O Oriented
P Programming

S1 = Python with Bianca

S2 = Java with Bryan

DT228(TU856)/DT282(TU858) - 2

COMPUTER SCIENCE

Testing

Objectives

- Discuss the importance of testing
- Discover unit testing
- Analyse test driven development

The Why!

- Think back at the programs we’ve written in the labs
- Did you write them first and then went back to fix all the ‘little’ issues?
- How many? How long did it take?
- How confident are you that you’ve found all the issues?

The What!

- Ensure code is working like you think it should
- Ensure code is still working when you’ve changed a small part of it
- Requirements are understood
- Maintenance aspect of code

Test Driven Development

Different way of thinking about the development lifecycle!

- “Write tests first”
  - Untested code is broken code, or
  - Only unwritten code should be tested
- Write the **test first**, then write the code to make sure that the test passes
- Allows us to focus on how the code will be interacted with
- Break up the problem into smaller parts
- The test becomes part of the design
- Allows you to discover anomalies in the design
- Makes for better code
  - **Opposite of banana software** – ripens at the customers’

Unit Testing

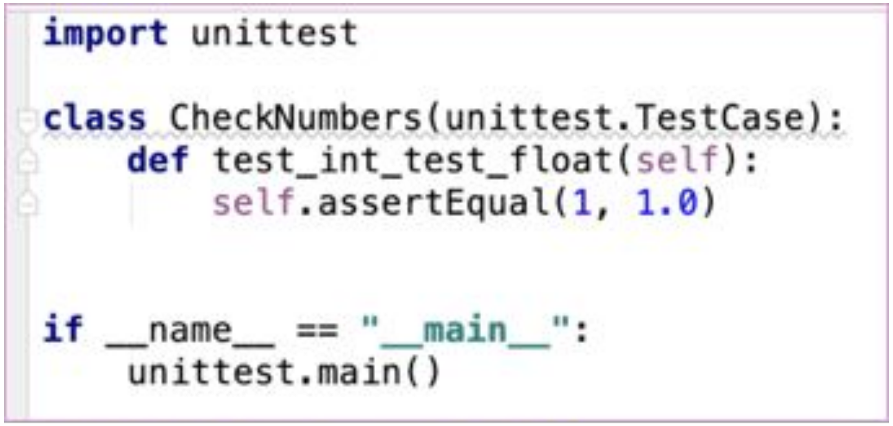
- Python built in test library
- Focus is on testing the least amount of code possible in one test
- Python library name: `unittest`
    - `TestCase` class
- Compare values
- Set up tests
- Clean up after tests have finished

Create a sub class to test a specific piece of Python code.

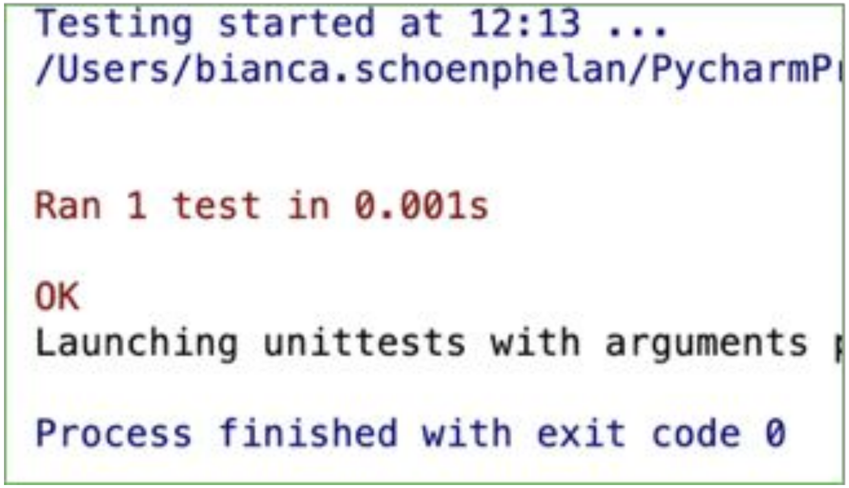
How it works

- Create a sub class of `TestCase`
- Every test method starts with `test`
  - Framework then runs the test
- Tests set some value of an object and then test and built in comparison methods to check that right results have been returned

Example



```
import unittest

class CheckNumbers(unittest.TestCase):
    def test_int_test_float(self):
        self.assertEqual(1, 1.0)


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

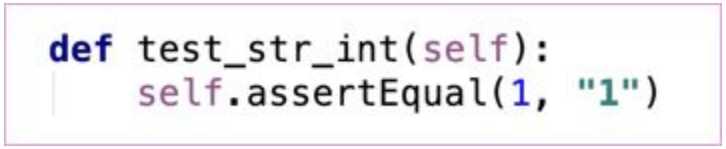


```
Testing started at 12:13 ...
/Users/bianca.schoenphelan/PycharmP


Ran 1 test in 0.001s

OK
Launching unittests with arguments

Process finished with exit code 0
```

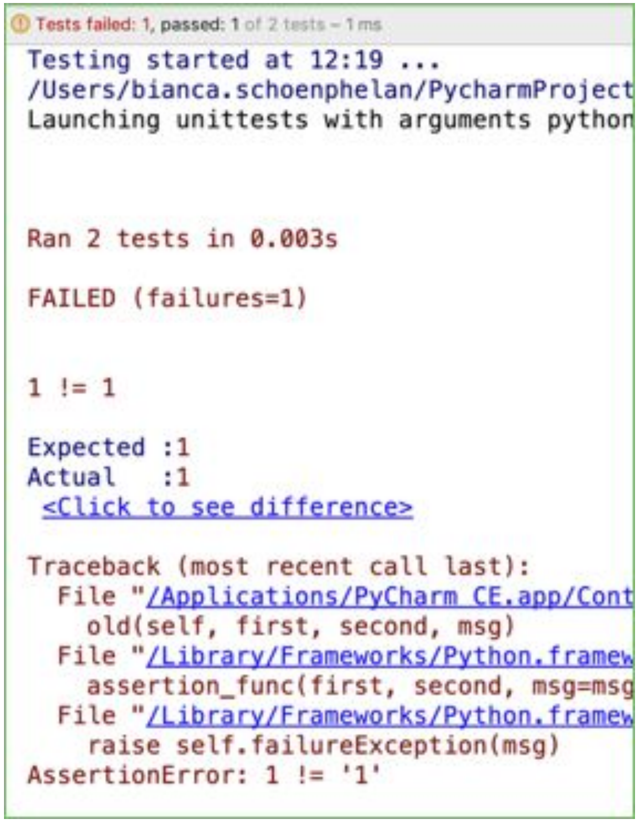
- First sub class from `TestCase`
- `Test_int_test_float` checks and either raises an exception or succeed depending on if the two parameters are equal
- Did you know that they compare as equals?

[1]

Example



```
def test_str_int(self):
    self.assertEqual(1, "1")
```



Tests failed: 1, passed: 1 of 2 tests – 1 ms

```
Testing started at 12:19 ...
/Users/bianca.schoenphelan/PycharmProject
Launching unittests with arguments python


Ran 2 tests in 0.003s

FAILED (failures=1)


1 != 1

Expected :1
Actual   :1
 <Click to see difference>

Traceback (most recent call last):
  File "/Applications/PyCharm CE.app/Cont
    old(self, first, second, msg)
  File "/Library/Frameworks/Python.framew
    assertion_func(first, second, msg=msg
  File "/Library/Frameworks/Python.framew
    raise self.failureException(msg)
AssertionError: 1 != '1'
```

- One test passes (the previously written one)
- One test failed
- Observe the fail exception
- We can have as many tests in one test case as we like
- All need to start with the word test, rest is handled automatically
- Results from one test should have no impact on other tests
- Write them as independent tests!

Good unit tests are kept as short as possible.

General Test Layout

1. Set certain variables to known values
2. Run one or more functions, methods, processes
3. Prove that expected results were returned by inspecting `TestCase` assertion method outputs

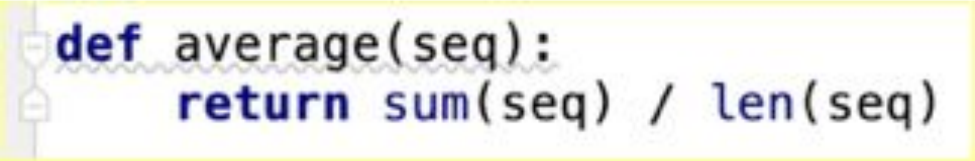
Assertion Methods

All assertion methods presented here take an optional msg attribute:

- It will be included in the error message if included
- Opportunity to insert some useful information for yourself or for whoever has to evaluate the errors.

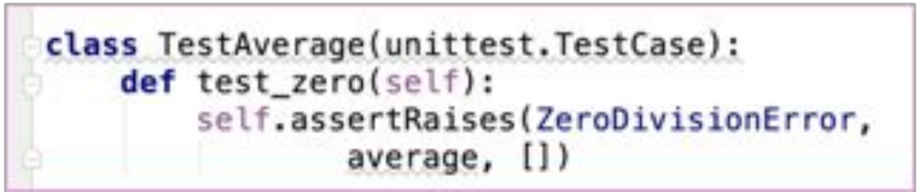
- `assertEqual`
  - Checks if two values are the same
- `assertNotEqual`
  - Does the opposite of above
- `assertTrue`/`assertFail`
  - Test an if statement
- `assertRaises`
  - Checks if a specific function call raises a specific exception, or context manager to wrap inline code

Example



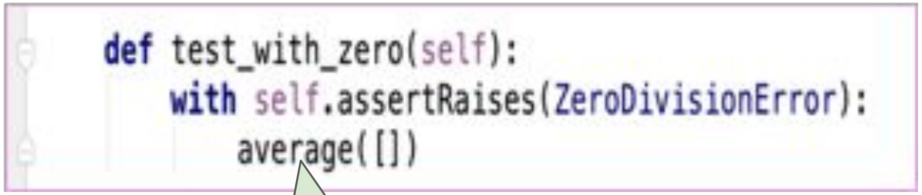
```python
def average(seq):
    return sum(seq) / len(seq)
```

Specific function calls exception



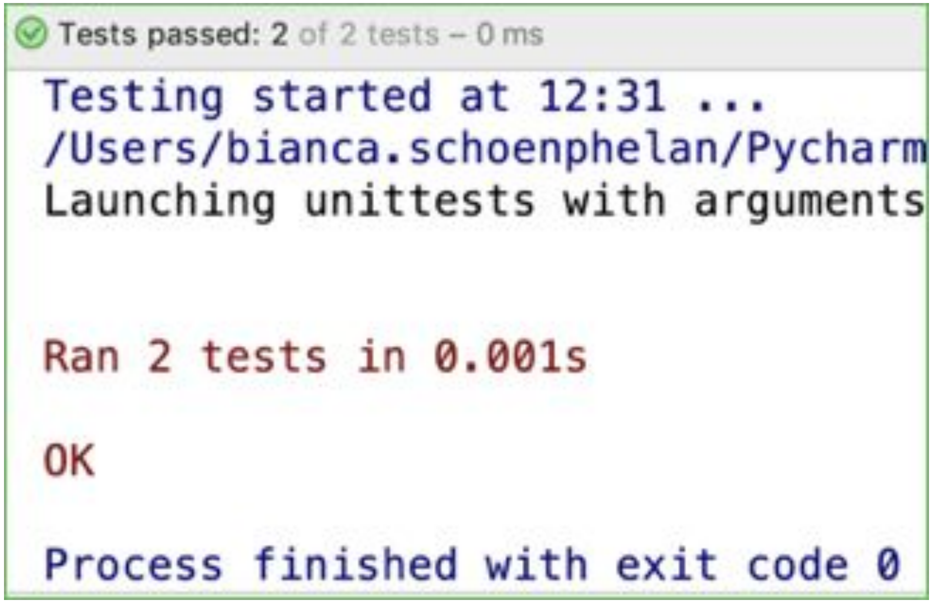
```python
class TestAverage(unittest.TestCase):
    def test_zero(self):
        self.assertRaises(ZeroDivisionError,
                          average, [])
```

Context manager with inline raising exception



```python
    def test_with_zero(self):
        with self.assertRaises(ZeroDivisionError):
            average([])
```

[1]



✓ Tests passed: 2 of 2 tests – 0 ms

```
Testing started at 12:31 ...
/Users/bianca.schoenphelan/Pycharm
Launching unittests with arguments


Ran 2 tests in 0.001s

OK

Process finished with exit code 0
```

Allows you to write code the way you normally would, by calling functions or executing code directly rather than having to wrap the function call into another function call

More Assertion Methods

- `assertGreater`
- `assertGreaterEqual`
- `assertLess`
- `assertLessEqual`

Take two comparable objects and make sure the desired inequality holds.

- `assertIn`
- `assertNotIn`

Ensures that an element is in or is not in a certain container.

Assertion Methods cont'd

- `assertIsNone`
- `assertIsNotNone`

Ensures that an element is in or is not None. But not any other false value.

- `assertSameElements`

Ensures two container objects have the same elements. Ignores the order.

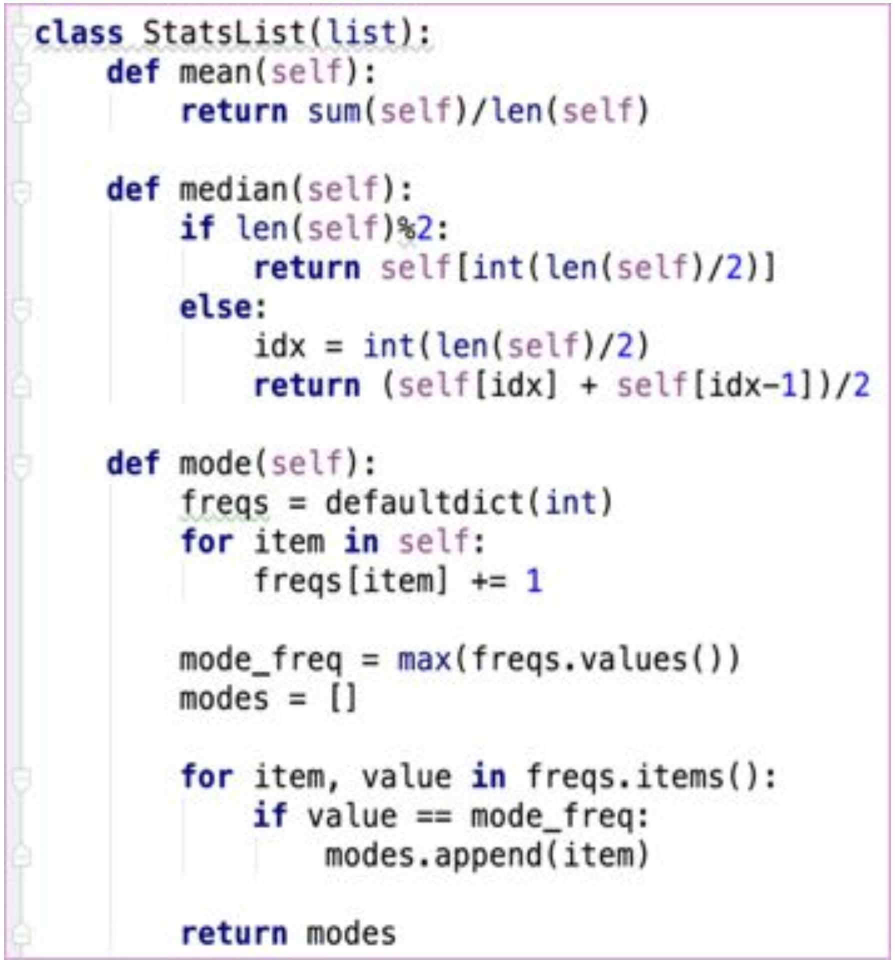
- `assertSequenceEqualassertDictEqual`
- `assertSetEqual`
- `assertListEqual`
- `assertTupleEqual`

Ensures two container objects have the same elements in the same order. If there is a difference you will get a diff displayed. Last four also test the type of list.

Reducing Efforts

- We often find that many test cases have the same set up
- For example, you may want to check calculation methods for correct outputs but use the same values for them
    - `setUp()` method lets you set up a container for all the elements that the following tests need

setUp()

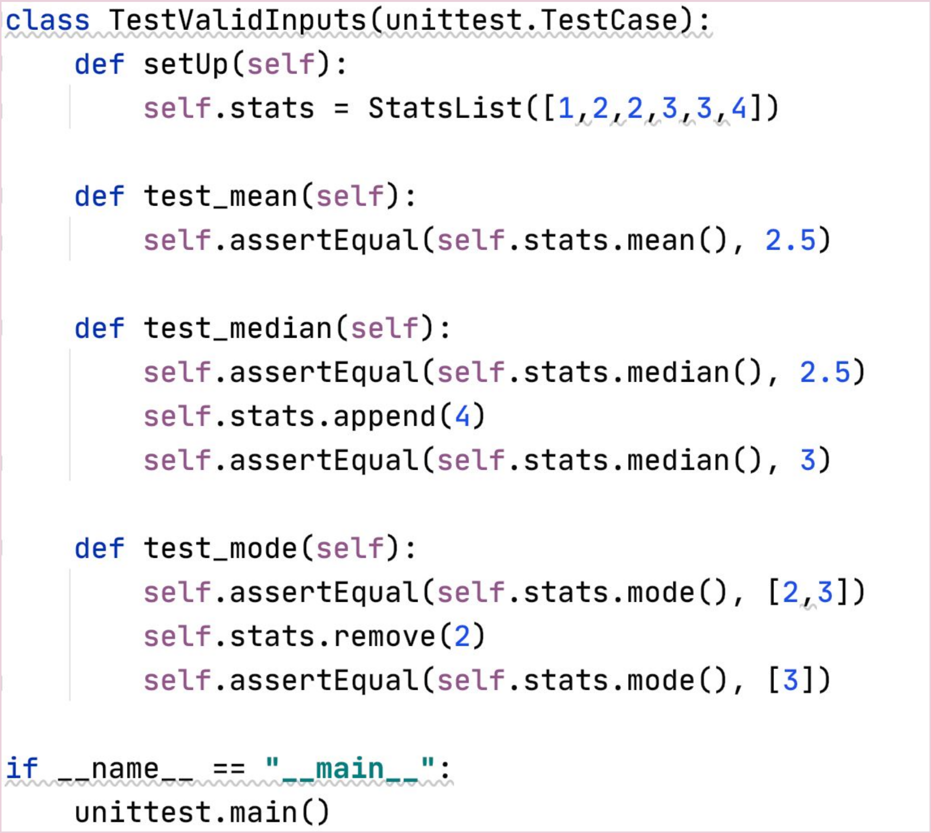


```
class StatsList(list):
    def mean(self):
        return sum(self)/len(self)

    def median(self):
        if len(self)%2:
            return self[int(len(self)/2)]
        else:
            idx = int(len(self)/2)
            return (self[idx] + self[idx-1])/2

    def mode(self):
        freqs = defaultdict(int)
        for item in self:
            freqs[item] += 1

        mode_freq = max(freqs.values())
        modes = []

        for item, value in freqs.items():
            if value == mode_freq:
                modes.append(item)

        return modes
```

- We will want to test similar situations for these methods:
  - What happens in an empty list,
  - What happens for lists with non-numeric values,
  - What happens for lists with normal data sets

[1]

setUp()



```python
class TestValidInputs(unittest.TestCase):
    def setUp(self):
        self.stats = StatsList([1,2,2,3,3,4])

    def test_mean(self):
        self.assertEqual(self.stats.mean(), 2.5)

    def test_median(self):
        self.assertEqual(self.stats.median(), 2.5)
        self.stats.append(4)
        self.assertEqual(self.stats.median(), 3)

    def test_mode(self):
        self.assertEqual(self.stats.mode(), [2,3])
        self.stats.remove(2)
        self.assertEqual(self.stats.mode(), [3])

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

- All 3 will pass
- We never explicitly call `setUp()`, the framework does that for us
- Observe how `test_median()` alters the list
- We return to previous list for `test_mode()`, otherwise we would have returned 3 values here because of the 2 fours

setUp() is called individually before each test.

Tests can be executed in any order and the result of one does not influence the other.

[1]

tearDown()

- Cleaning up after a test has run
- Useful if we do anything else but work on objects
- Example: file I/O, database connections, test created files
- Ensures that the system is in the same state before test is run

Groups tests together in sub classes by set up and clean up required.

Organising Your Tests

- Goal:
    - Trivial to run
    - Quick yes/no style answers to ‘did my recent change break anything’
- If you have many unit tests it can be easy to lose sight of what you want to execute and in what order discover module solves this
    - Looks for any modules in the current folder or subfolders starting with test
    - To use make sure your modules are called test_something.py and then run with

    -
      ```
      python3 – m unitttest discover
      ```

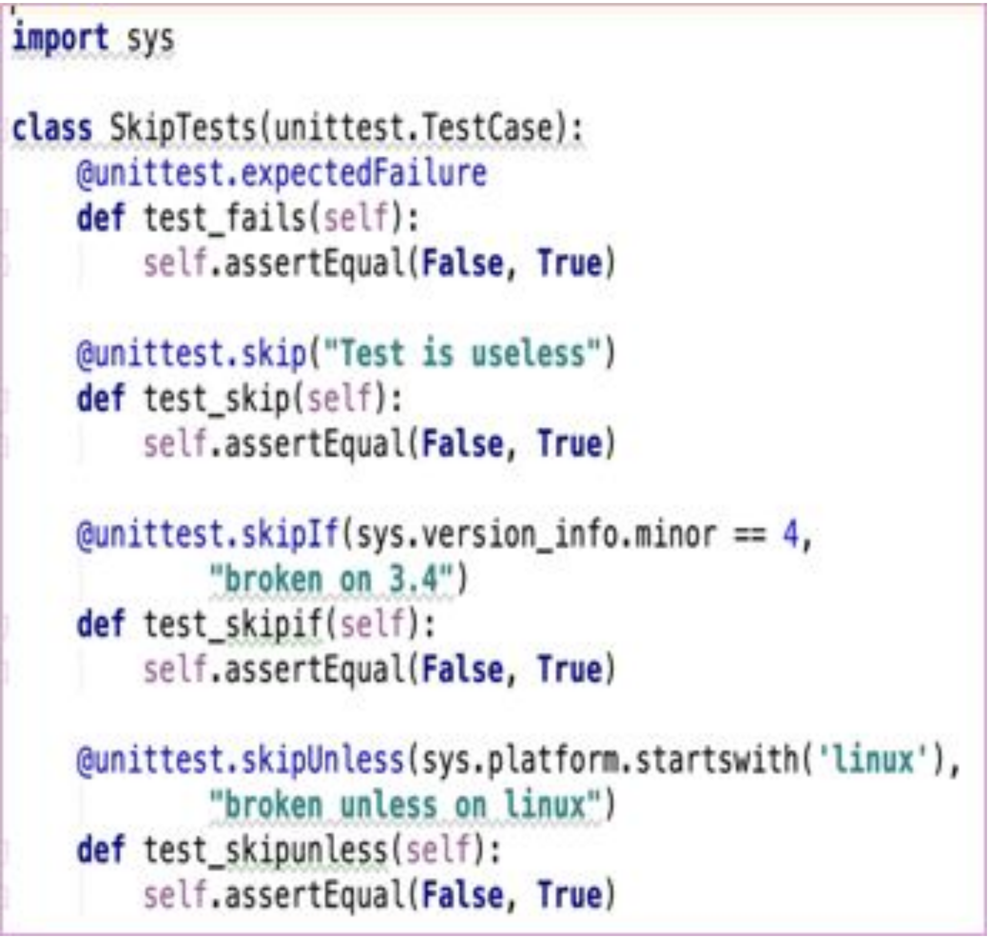
Ignore Broken Tests

- Maybe feature isn't finished yet
- Or feature is only available on a certain platform, Python version or advanced library version
- Use decorators in this case to indicate that a test is expected to fail or to skip over it

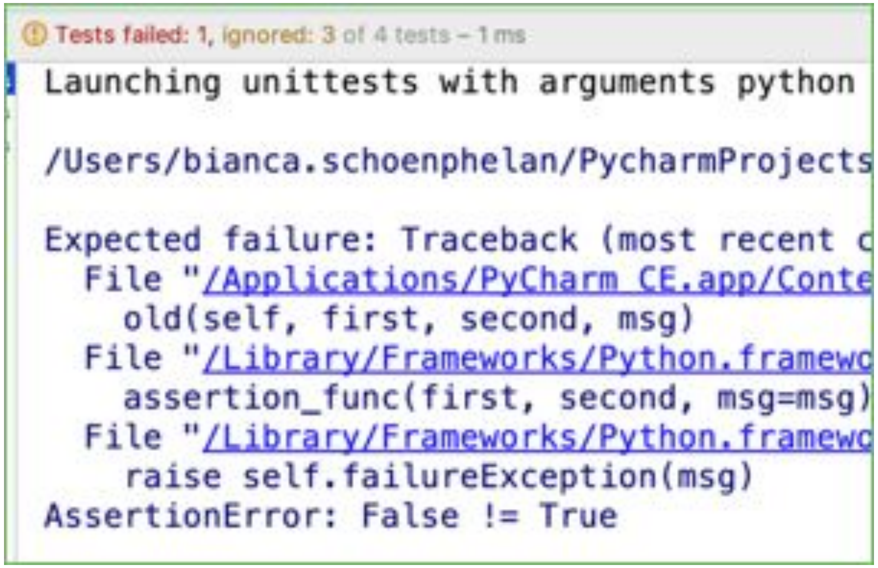
- `expectFailure()`
- `skip(reason)`
- `skipIf(condition, reason)`
- `skipUnless(condition,reason)`

[1]

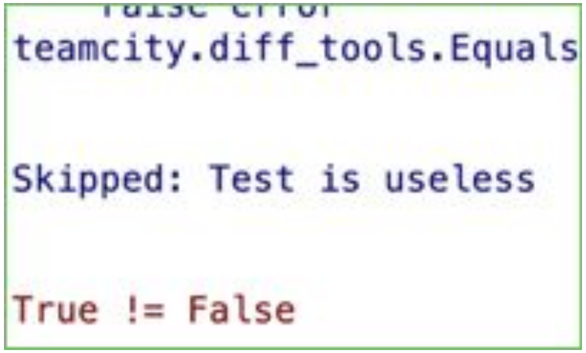
Example



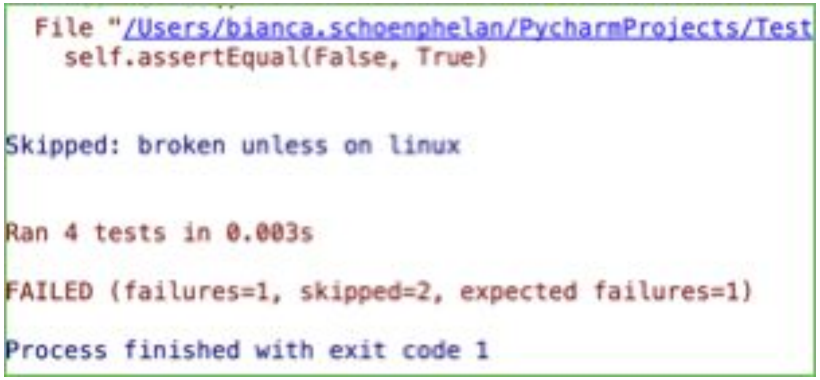
```
import sys

class SkipTests(unittest.TestCase):
    @unittest.expectedFailure
    def test_fails(self):
        self.assertEqual(False, True)

    @unittest.skip("Test is useless")
    def test_skip(self):
        self.assertEqual(False, True)

    @unittest.skipIf(sys.version_info.minor == 4,
            "broken on 3.4")
    def test_skipif(self):
        self.assertEqual(False, True)

    @unittest.skipUnless(sys.platform.startswith('linux'),
            "broken unless on linux")
    def test_skipunless(self):
        self.assertEqual(False, True)
```



```
Tests failed: 1, ignored: 3 of 4 tests – 1 ms
Launching unittests with arguments python

/Users/bianca.schoenphelan/PycharmProjects

Expected failure: Traceback (most recent c
  File "/Applications/PyCharm CE.app/Conte
    old(self, first, second, msg)
  File "/Library/Frameworks/Python.framewo
    assertion_func(first, second, msg=msg)
  File "/Library/Frameworks/Python.framewo
    raise self.failureException(msg)
AssertionError: False != True
```



```
raise error
teamcity.diff_tools.Equals


Skipped: Test is useless


True != False
```



```
  File "/Users/bianca.schoenphelan/PycharmProjects/Test
    self.assertEqual(False, True)


Skipped: broken unless on linux


Ran 4 tests in 0.003s

FAILED (failures=1, skipped=2, expected failures=1)

Process finished with exit code 1
```

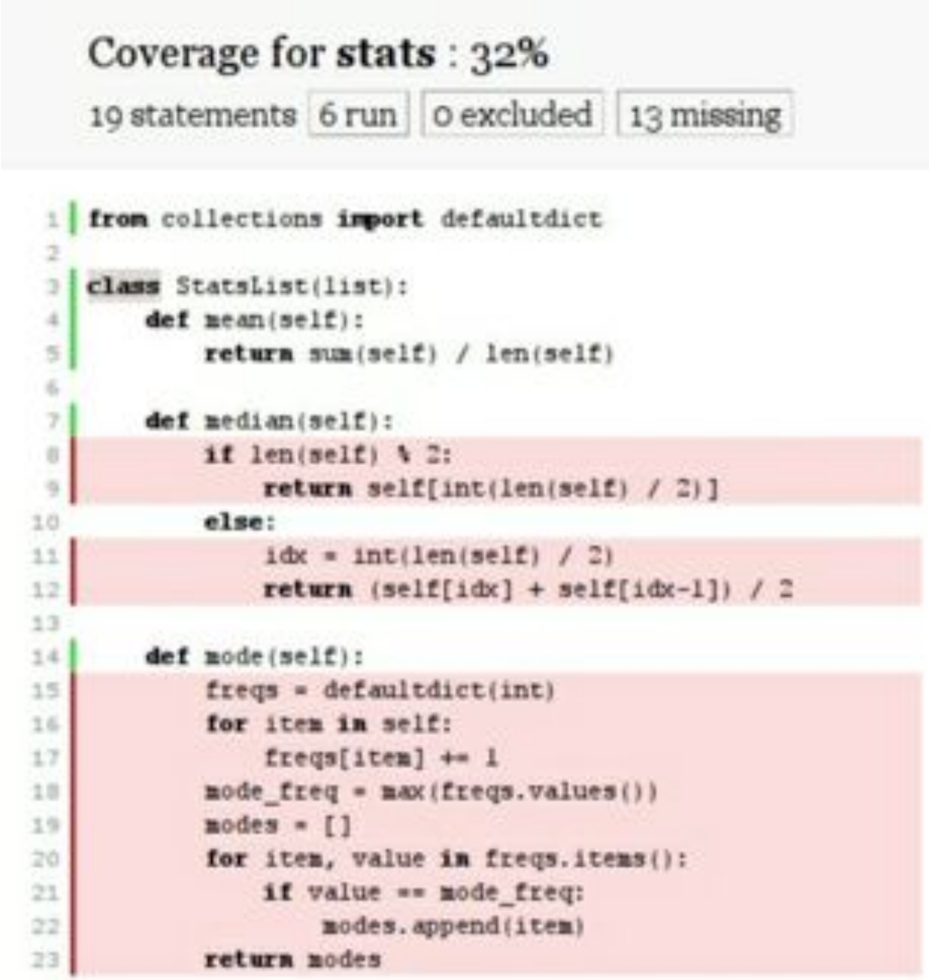
Other Options

- There are different testing frameworks, for example Pytest
- Check which ones Pycharm supports
  - If you want a different one you need to switch it in the preferences in Pycharm as unittest is the default

How much Testing is Enough?

- Untested code is broken code
  - But how much testing is enough testing?
- Very difficult to answer
  - Even if we test every line in our code it’s hard to be sure
  - Code coverage might be excellent but still are we testing the right thing?
  - Code coverage tested with coverage.py
- `pip install coverage`
  - Gives a coverage report

Coverage Output



Coverage for **stats** : 32%

19 statements | 6 run | 0 excluded | 13 missing

```
from collections import defaultdict

class StatsList(list):
    def mean(self):
        return sum(self) / len(self)

    def median(self):
        if len(self) % 2:
            return self[int(len(self) / 2)]
        else:
            idx = int(len(self) / 2)
            return (self[idx] + self[idx-1]) / 2

    def mode(self):
        freqs = defaultdict(int)
        for item in self:
            freqs[item] += 1
        mode_freq = max(freqs.values())
        modes = []
        for item, value in freqs.items():
            if value == mode_freq:
                modes.append(item)
        return modes
```

[1], p. 384

Test Driven Development

Purpose is to make you think about every feature and every section of code individually.

1. Write the test
   - Create a new test
   - Should be a simple tes, succinct and tests every aspect of a bigger program
2. Confirm the test fails
   - No code yet, so test should fail
3. Write code to pass the test
   - Write code to pass the test
   - No added code, just for the test
4. Confirm the test passes
5. Refactor
   - Duplications or ambiguities might have been introduced in previous steps
   - Locate problem areas and simplify the code
6. Repeat all steps
   - We start off very small and then increase to our full fledged featured software system

Code here is typically quite rough and unfinished. That's ok!

[2]

Pros and Cons of TDD

Advantages

- Less reliance on debugging because we start with the test first and then write the code
- More user focused as the brain has to work backwards from the test to the code
- May decrease overall development time although total lines of code typically increase. Prevents and catches bugs early.

Disadvantages

- No big picture design anymore as we start with the simplest test and then work up, so we might miss the forest for the trees
- Good for small projects or features, unwieldy to apply to massive projects
- Significant upfront energy on testing

Summary

- ★ **Unit Testing in Python**
- ★ **Test Driven Development**

References

1. Python 3: Object Oriented Programming, Dusty Phillips, 2nd edition, 2015
2. Test driven development, Andrew Powell-Morse, 11/04/2017, https://airbrake.io/blog/sdlc/test-driven-development, accessed Nov 2019.