More Assertion Types

assertStuff()

• assertFalse • assertTrue • assertRaises • assertWarns • assertLogs • assertEqual • assertNotEqual • assertAlmostEqual • assertNotAlmostEqual • assertSequenceEqual • assertListEqual	• assertTupleEqual • assertSetEqual • assertIn • assertNotIn • assertIs • assertIsNot • assertDictEqual • assertDictContainsSubset • assertCountEqual • assertMultiLineEqual • assertLess	• assertLessEqual • assertGreater • assertGreaterEqual • assertIsNone • assertIsNotNone • assertIsInstance • assertNotIsInstance • assertRaisesRegex • assertWarnsRegex • assertRegex • assertNotRegex

Expecting Exceptions

Sometimes your code is *supposed* to raise an exception. And it's an error if, in that situation, it does not.

Use `TestCase.assertRaises()` to verify.

Imagine a `roman2int()` function:

```
>>> roman2int("XVI")
16
>>> roman2int("II")
2
>>> roman2int("a thousand")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in roman2int
ValueError: Not a roman numeral: a thousand
```

Asserting Exceptions

```
import unittest
from roman import roman2int

class TestRoman(unittest.TestCase):
    def test_roman2int_error(self):
        with self.assertRaises(ValueError):
            roman2int("bad value")
```

Catching The Error

If `roman2int()` does NOT raise `ValueError`:

```
$ python3 -m unittest test_roman2int.py
F
======================================================
FAIL: test_roman2int_error (test_roman2int.TestRoman)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_roman2int.py", line 7, in test_roman2int_error
    roman2int("bad value")
AssertionError: ValueError not raised

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Pinpointing

```
import unittest
from roman import roman2int

class TestRoman(unittest.TestCase):
    def test_roman2int_error(self):
        with self.assertRaises(ValueError):
            roman2int("bad value")
```

Inspecting Exceptions

You can also make assertions on the exception object itself. To do this, capture its *context* with an "as" clause:

```
import unittest
from roman import roman2int

class TestRoman(unittest.TestCase):
    def test_roman2int_error(self):
        with self.assertRaises(ValueError) as context:
            roman2int("bad value")
        exception = context.exception
        expected_message = "Not a roman numeral: bad value"
        actual_message = exception.args[0]
        self.assertEqual(expected_message, actual_message)
```

assertIn

`assertIn()` asserts an element is in a collection.

`assertNotIn()` asserts it's not.

```
numbers = [1, 3, 5, 7, 9]
self.assertIn(3, numbers) # passes
self.assertIn(2, numbers) # fails
self.assertNotIn(2, numbers) # passes
```

`assertIn(x, items)` is equivalent to `assertTrue(x in items)`.

assertAlmostEqual

Checks whether two numbers are approximately equal.
Specify how close with `places` or `delta`.

```
from mymathlib import newton_roots
# newton_roots() uses Newton's Method to find
# a root of an equation.
expected = 1.4142135623730951 # square root of 2
# coefficients for "x**2 - 2"
coefficients = (1, 0, -2)
# Calculate root of "x**2 - 2 == 0", starting search at x == 0.1
actual = newton_roots(coefficients, 0.1)
# check for almost-equality, with default tolerance
self.assertAlmostEqual(expected, actual)
# Check with lower precision
self.assertAlmostEqual(expected, actual, places=3)
# Check with higher precision
self.assertAlmostEqual(expected, actual, places=10)
# Check by absolute difference
self.assertAlmostEqual(expected, actual, delta=.0001)
```

Also available: `assertNotAlmostEqual()`.

assertIs

`assertIs()` asserts on identity:

```
# True if and only if id(x) == id(y)
self.assertIs(x, y)
# Equivalent to:
self.assertTrue(x is y)
```

`assertIsNone()` is just a more specialized version:

```
self.assertIsNone(x)
# Equivalent to these two:
self.assertIs(None, x)
self.assertTrue(x is None)
```

Also available: `assertIsNot()` and `assertIsNotNone()`.

assertIsInstance

`assertIsInstance()` verifies that an object is an instance of a certain type, or its superclass.

We'll use this hierarchy to demonstrate:

```
# For a financial security
class Security:
    def __init__(self, symbol):
        # ....
class Stock(Security):
    # ....
class Derivative(Security):
    # ...
class CallOption(Derivative):
    def __init__(self, underlying, strike):
        # ...

my_stock = Stock("MSFT")
my_option = CallOption("MSFT", 100)
```

assertIsInstance

Format is `assertIsinstance(some_object, TypeName)`:

```
self.assertIsinstance(my_stock, Stock) # passes
self.assertIsinstance(my_option, Stock) # fails
self.assertIsinstance(my_option, CallOption) # passes
self.assertIsinstance(my_option, Security) # passes
self.assertIsinstance(my_option, Derivative) # passes
self.assertIsinstance(my_stock, Security) # passes
self.assertIsinstance(my_stock, Derivative) # fails
```

`assertIsinstance(x, klass)` is equivalent to `assertTrue(instance(x, klass))`. Also available: `assertNotIsInstance()`.

assert < <= => >

```
x = 1.0
y = 1.5
z = 1.5

self.assertLess(x, y) # passes
self.assertLess(y, z) # fails
self.assertLessEqual(y, z) # passes
self.assertGreater(x, y) # fails
self.assertGreater(y, x) # passes
self.assertGreaterEqual(y, z) # passes
```

`assertLess(x, y)` is equivalent to `assertTrue(x < y)`, and so on.

Works for anything comparable (e.g. strings), not just numbers.

assertRegex

`assertRegex()` asserts that a string matches a regular expression.

```
# Check that string is a date like "2034-12-01"
pattern = r"^{\d}4-{\d}2-{\d}2$"
value = client.get_date()
self.assertRegex(value, pattern)
# Equivalent to:
import re
match = re.search(pattern, value)
self.assertIsNotNone(match)
```

Note the order of args is reversed for `assertRegex()` and `re.search()`.

Also available: `assertNotRegex()`

Lab: Intermediate Unit Tests

Instructions: `lab-intermediate.txt`

(Note this builds on the previous lab - you must complete it first.)

- In `labs` folder
- First follow the instructions to modify `textlib.py` and `test_textlib.py`
- When you are done, study the solution - compare to what you wrote.
- ... and then optionally follow the extra credit instructions