UNIT TESTING

CS 3030: Python
Instructor: Damià Fuentes Escoté

University of Colorado
Colorado Springs

Quiz 5

UNITTEST MODULE

unittest module

- The unittest module provides a rich set of tools for constructing and running tests.
- Unit Testing is the first level of software testing where the smallest testable parts of a software are tested. This is used to validate that each unit of the software performs as designed.
- More info:
  - *https://docs.python.org/3/library/unittest.html*

For what?

```
def is_palindrome(var):
    assert isinstance(var, str), 'The input should be a string’
    namesRegex = re.compile(r'''[^a-zA-Z]'’’)
    var = namesRegex.sub('', var)
    var = var.lower()
    return var == var[::-1]
```

For what?

```
def is_palindrome(var):
    assert isinstance(var, str), 'The input should be a string’
    namesRegex = re.compile(r'''[^a-zA-Z]'’’)
    var = namesRegex.sub('', var)
    var = var.lower()
    return var == var[::-1]
```

- Unit tests that should return **True**:
  - *“madam”, “race car”, “Dammit, I'm Mad!”…*
- Unit tests that should return **False**:
  - *“hello world”, “Alice”…*
- Unit tests that should raise an **error**:
  - *2*

For what?

```
def is_palindrome(var):
    assert isinstance(var, str), 'The input should be a string’
    namesRegex = re.compile(r'''[^a-zA-Z]'’’)
    var = namesRegex.sub('', var)
    var = var.lower()
    return var == var[::-1]
```

- This way, anyone that modifies the function will run the tests again to check that everything is working correctly.
- If the tests are not passed, means that something is wrong and you have to modify it again.

unittest module

- A testcase is created by subclassing unittest.TestCase.

```
import unittest

class TestStringMethods(unittest.TestCase):
    # Individual tests here
```

unittest module

- The three individual tests are defined with methods whose names start with the letters **test**. This naming convention informs the test runner about which methods represent tests.
- The crux of each test is a call to **assertEqual()** to check for an expected result; **assertTrue()** or **assertFalse()** to verify a condition; or **assertRaises()** to verify that a specific exception gets raised. These methods are used instead of the assert statement so the test runner can accumulate all test results and produce a report.
- The **setUp()** and **tearDown()** methods allow you to define instructions that will be executed before and after each test method.

unittest module

```
def test_upper(self):
    self.assertEqual('foo'.upper(), 'FOO’)

def test_isupper(self):
    self.assertTrue('FOO'.isupper())
    self.assertFalse('Foo'.isupper())

def test_split(self):
    s = 'hello world’
    self.assertEqual(s.split(), ['hello', 'world’])
    # check that s.split fails when the separator is not a string
    with self.assertRaises(TypeError):
        s.split(2)
```

unittest module

- The final block shows a simple way to run the tests. unittest.main() provides a command-line interface to the test script.

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

Output:

```
Ran 3 tests in 0.000s
OK
```

unittest module

- The **setUp()** and **tearDown()** methods allow you to define instructions that will be executed before and after each test method.

```python
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def getCompleteName(self):
        return f"{self.name} {self.surname}"
```

unittest module

- The **setUp()** and **tearDown()** methods allow you to define instructions that will be executed before and after each test method.

```
class PersonTestCase(unittest.TestCase):
    def test_complete_name(self):
        person = Person('Bob','Miller’)
        self.assertEqual(person.getCompleteName(), "Bob Miller")

    def test_name(self):
        person = Person('Bob','Miller’)
        self.assertEqual(person.name, "Bob")
```

unittest module

- The setUp() and tearDown() methods allow you to define instructions that will be executed before and after each test method.

```
class PersonTestCase(unittest.TestCase):
    def setUp(self):
        self.person = Person('Bob','Miller’)

    def test_complete_name(self):
        self.assertEqual(self.person.getCompleteName(), "Bob Miller")

    def test_name(self):
        self.assertEqual(self.person.name, "Bob")
```

TDD – Test Driven Development

- Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements.
- Requirements → Code the tests → Code the software to meet the tests
- You don’t need to do TDD to write unit tests!!

https://en.wikipedia.org/wiki/Test-driven_development

Are unit tests worth the effort?

- Unit Tests allows you to make big changes to code quickly. You know it works now because you've run the tests, when you make the changes you need to make, you need to get the tests working again.
  - *Although you can be very lazy, doing them saves a LOT of hours.*

https://stackoverflow.com/questions/67299/is-unit-testing-worth-the-effort

Are unit tests worth the effort?

- TDD helps you to realize when to stop coding. Your tests give you confidence that you've done enough for now and can stop tweaking and move on to the next thing.

https://stackoverflow.com/questions/67299/is-unit-testing-worth-the-effort

Are unit tests worth the effort?

- Unit Tests help you really understand the design of the code you are working on. Instead of writing code to do something, you are starting by outlining all the conditions you are subjecting the code to and what outputs you'd expect from that.

https://stackoverflow.com/questions/67299/is-unit-testing-worth-the-effort

Are unit tests worth the effort?

- Contrary to popular belief unit testing does not mean writing twice as much code, or coding slower. It's faster and more robust than coding without tests once you've got the hang of it. Test code itself is usually relatively trivial and doesn't add a big overhead to what you're doing. This is one you'll only believe when you're doing it :)

https://stackoverflow.com/questions/67299/is-unit-testing-worth-the-effort