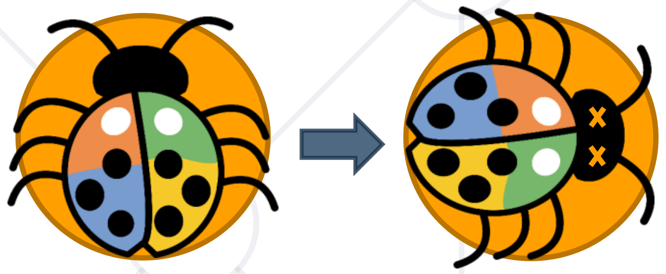
Unit Testing

Building Rock-Solid Software



SoftUni Team

Technical Trainers

Software University

https://softuni.bg

Table of Contents

Questions

Software University

sli.do

#python-advanced

Testing

What is Testing

What is Testing?

- The first level of **software testing**
  - The smallest **testable** parts of a software are tested
- Validates that each unit of the software **performs as designed**
- Types of testing:
  - **Manual** testing
  - **Automated** testing
    - **Unit** testing
    - **Integration** testing
    - Many more types of testing

What is manual testing?

- Manually test the code as a standard user
  - Go to each page of a web application
  - Test every behavior and functionality
- And this happens every time
  - A new feature is introduced
  - A bug is fixed
  - A requirement is changed

Drawbacks from Manual Testing

- Not **repeatable**
  - Automatically. Changing part of the code
- Hard to **structure**
  - Depends on the manual tester
- Less **accuracy**
  - The possibility of "human error" is applicable here
- **Not** as **easy** as it should be
- Requires more **time** and **resources**

Automated testing (1)

- Automated testing represents business requirements in code
  - i.e., code that verifies code
- Types of automated tests
  - Unit tests
  - Integration tests
  - Functional/UI/E2E tests
  - System tests
  - Regression tests
  - etc..

Automated testing (2)

Software University

- Done through an **automation tool**
- Higher **accuracy**
- Better **reporting capabilities**
- Increased **coverage**
- Improved **bug detection**
- Increased **reusability**
- Stability

Benefits of automated testing

- Automated tests:
  - are automatically repeatable
  - fail as early as possible
  - enable the presentation of business requirements in code
  - reduce the **cost of change**
  - **decrease** the number of **defects** in the code
- Bonus:
  - Improve **design**

Code conventions while testing

- While writing tests, different conventions and practices are used
  - **Less** abstract, **more** concrete
  - Test **specific** cases
- **Triple A** pattern:
  - Arrange
  - Act
  - Assert

What is Unit Testing?

What is Unit Testing?

- **Unit Testing** is a type of software testing where individual units or components of a software are tested
- The purpose is to validate that each unit of the software code **performs as expected**
- Unit Testing is done **during the development** (coding phase) of an application by the developers

Unit Testing Framework

Unit Testing Framework

- Individual **units** or **components** are being tested
- Validate **each unit** to perform as expected
- A unit may be an **individual**:
  - Function
  - Method
  - Procedure
  - Modules
  - Object

Concepts Behind `unittest` (1)

- Test fixture
  - A **baseline** for running tests to ensure there is a **fixed environment** in which tests are run so that results are **repeatable**
- Test case
  - A **set of conditions** used to determine if a system works **correctly**
- Test suite
  - A **collection** of **`testcases`** used to test software if it has some specified set of behaviors

Concepts Behind unittest (2)

- Test runner
  - A component that **sets up the execution** of tests and provides the **outcome** to the user

```
import unittest
class SimpleTest(unittest.TestCase):
    def test_upper(self):
        result = 'foo'.upper()
        expected_result = 'FOO'
        self.assertEqual(result, expected_result)

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

Running the Tests (1)

- Run by the following block of code

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

- Results printed on the console

```
----------------------

Ran 1 test in 0.00s

OK
```

Test outcome

Running the Tests (2)

- The possible outcomes are
  - OK – **all** tests **passed**
  - FAIL – **one or many** tests **failed**, and an **`AssertionError`** exception is raised
  - ERROR – the tests raised an exception **other than `AssertionError`**

Basic Unittest Terms (1)

- **`unittest.TestCase`** – create test cases by **subclassing** it
- **`assertEqual() / assertNotEqual()`** – tests that the two arguments are **equal**/**unequal** in value
- **`assertTrue() / assertFalse()`** – tests that the argument has a Boolean value of **True**/**False**
- **`assertIn() / assertNotIn()`** – tests that the first argument **is in** / **is not in** the second

Basic Unittest Terms (2)

- **`assertRaises()`** – **raises** a specific **exception**
- **`unittest.main()`** – provides a command-line **interface** to the test script
- **`setUp()`** – prepares the **test fixture**
  - The method is called **immediately before** the test method

Test Example (1)

- If we have a class Person with methods **get_full_name()** and **get_info()**:

```python
class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def get_full_name(self):
        return f'{self.first_name} {self.last_name}'

    def get_info(self):
        return f'{self.first_name} {self.last_name} is {self.age} years old'
```

Test Example (2)

- We can test both methods using the code below:

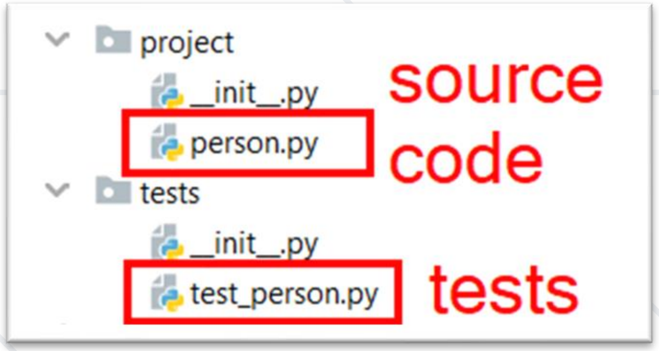
```python
import unittest

class PersonTests(unittest.TestCase):
    def setUp(self):
        self.person = Person("Luc", "Peterson", 25)

    def test_get_full_name(self):
        result = self.person.get_full_name()
        expected_result = "Luc Peterson"
        self.assertEqual(result, expected_result)

    def test_get_info(self):
        result = self.person.get_info()
        expected_result = "Luc Peterson is 25 years old"
        self.assertEqual(result, expected_result)

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

Unittest Modules

- **Advantages** to placing the test code in a **separate module**:
  - The test module can be run standalone from the **command line**
  - The test code can more **easily be separated** from the shipped code
  - Tested code can be **refactored** more easily
  - If the testing strategy changes, there is **no need** to **change the source code**

Unittest Modules Example

- Testing the `class Person` from the previous example:
  - Create the tests in a separate module



  - Include them in a package in order to be able to make proper imports from the modules

```
import unittest
from project.person import Person
```

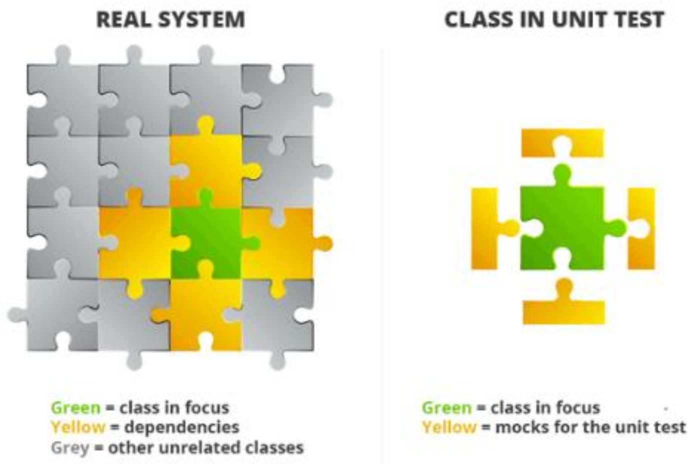
Mocking

What is Mocking?

- In plain English, mocking means "making a **replica** or **imitation** of something"
- Mocking is the way to test benefiting from isolation
  - **isolate** related logic into SRP modules
  - **simulate** the behavior of these modules

Mocking Example

- In unit testing, we want to test methods of one class in **isolation**, but classes are **not isolated**
- They are using **services** and **methods** from other classes
- We mock the services and methods from other classes and simulate the real behavior



Mocking in Python

- To use mocking in python, the built-in way is **unittest.mock**:

```python
@patch('app.hotel.RoomsManager')
def test_rent_room__when_no_free_rooms__should_raise(self, mock):
    RoomsManagerMock = mock.return_value
    RoomsManagerMock.has_free_rooms.return_value = False

    hotel = Hotel('At Joe\'s', 3, 2, 1)

    with self.assertRaises(NoFreeRoomError) as context:
        hotel.rent_room([], RoomTypes.APARTMENT)
    self.assertIsNotNone(context.exception)
```

How to Write Good Tests

Unit Testing Best Practices

Assertion Messages

- Assertions can **show messages**
  - Helps with **diagnostics**

```
def test_get_info(self):
    result = self.person.get_info()
    expected_result = "Luc Peterson is 25 years old"
    self.assertEqual(result, expected_result)
```

Assertion message

Naming Tests

- Test names
  - Should use **business domain terminology**
  - Should be **descriptive** and **readable**

✗

```
test_increment_Number(self): …
test_Test1(self): …
testTransfer(self): …
```

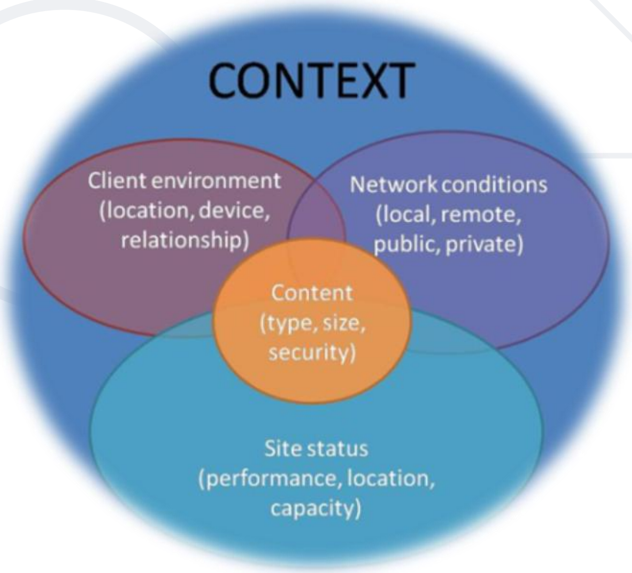
✓

```
test_deposit_Xleva_should_increase_balance_with_Xleva(self): …
test_deposit_negativeLeva__should_not_increase_balance(self): …
```

Seven Testing Principles

Seven Testing Principles (1)

- Testing is context dependent
  - Testing is done differently in **different contexts**
- Example:
  - Safety-critical software is tested **differently** from an e-commerce site

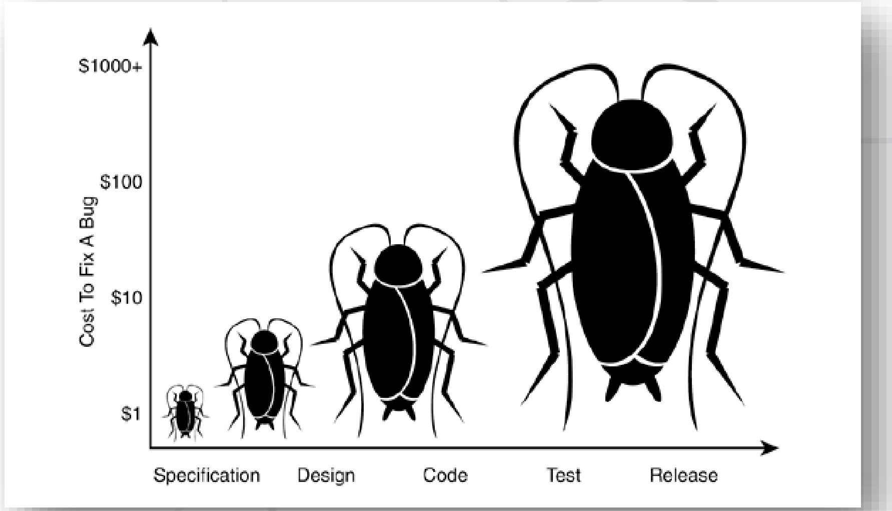

CONTEXT
Client environment (location, device, relationship)
Network conditions (local, remote, public, private)
Content (type, size, security)
Site status (performance, location, capacity)

Seven Testing Principles (2)

- Exhaustive testing is **impossible**
  - All combinations of inputs and preconditions are usually an almost **infinite number**
  - Testing everything is not feasible
    - Except for trivial cases
  - Risk analysis and priorities should be used to focus on testing efforts

Seven Testing Principles (3)

- Early testing is **always preferred**
  - Testing activities shall be started as early as possible
    - And shall be focused on defined objectives
  - The later a bug is found – the more it costs!



Seven Testing Principles (4)

- Defect clustering
  - Testing effort shall be focused **proportionally**
    - To the expected and later observed defect density of modules
  - A **small number** of modules usually contains **most of the defects** discovered
    - Responsible for most of the operational failures

Seven Testing Principles (5)

- Pesticide paradox
  - Same tests repeated **over and over again** tend to **lose their effectiveness**
  - Previously **undetected** defects remain **undiscovered**
  - New and modified test cases should be developed

Seven Testing Principles (6)

- Testing shows the presence of defects
  - Testing can **show that defects are present**
  - Cannot prove that there are no defects
  - Appropriate testing **reduces** the probability for defects

Seven Testing Principles (7)

- Absence-of-errors fallacy
  - **Finding** and **fixing** defects itself does not help in these cases:
    - The system built is unusable
    - Does not fulfill the user needs and expectations

Summary

- **Unit Testing** helps us build solid code
- **Structure** your unit tests – **3A Pattern**
- Use different **assertions** depending on the situation
- Concepts behind the **unittest framework**

Software University

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

createX

Postbank
Решения за твоето утре

BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

CODE IT UP

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License