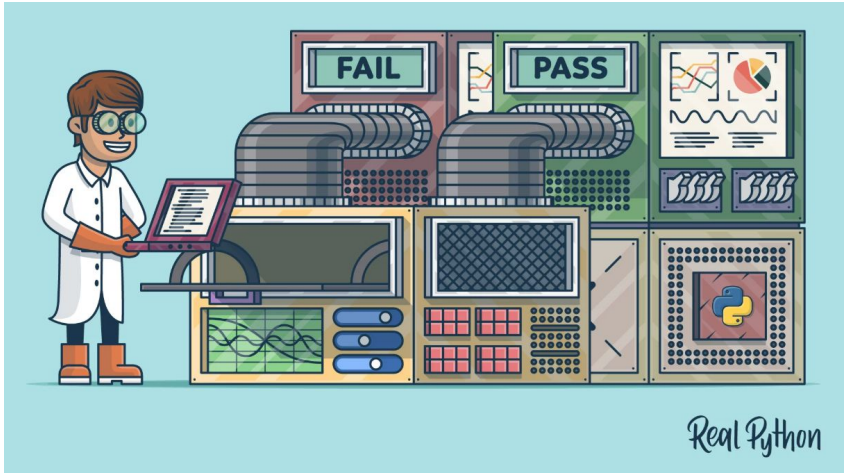
Test Driven Development & Python Unit Tests

IN104

Natalia Díaz Rodríguez, ENSTA ParisTech

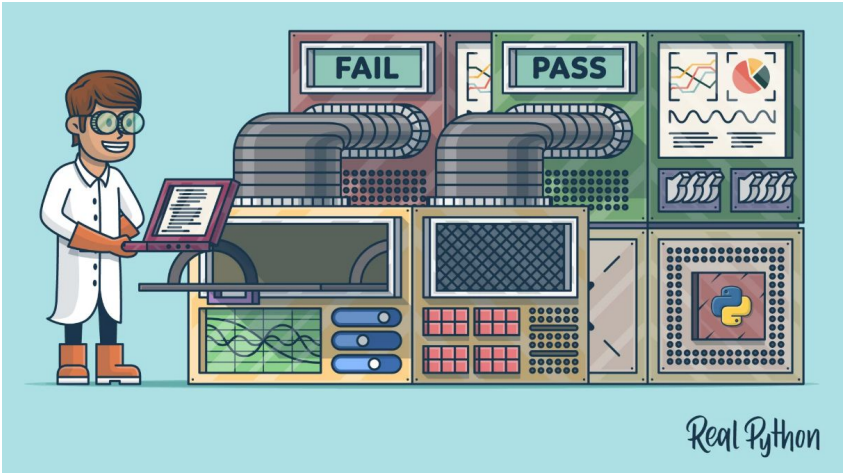
Testing frameworks

- **Unit Testing:** Test-Driven Development, Test-First Programming Philosophy
- **Regression Testing**: Collects all unit tests for individual modules into one big test suite, and runs them all at once.
    - 1st thing my automated build script does: Make sure all the examples still work
    - If fails: build immediately stops



TDD Cycle:

1. Add a little test
2. Run all tests and fail
3. Make a little change
4. Run the tests and succeed
5. Refactor to remove duplication



Unit Testing Philosophy:

Design and write test cases:

- before the code they are testing
- that test:
  - Good input and check for proper results
  - Bad input and check for proper failures
- to illustrate bugs or reflect new requirements
- to improve performance, scalability, readability, maintainability, or whatever other −ility you're lacking
- that are specific, automated, and independent

Unit Testing Philosophy:

Be comfortable:

- Subclassing **unittest.TestCase** and writing methods for individual test cases
- Using **assertEqual** to check that a function returns a known value
- Using **assertRaises** to check that a function raises a known exception
- Running unit tests in verbose [**-v**] or regular mode
- Calling **unittest.main()** in your **if __name__** clause to run all your test cases at once

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

Dependency and Duplication

- **Dependency**: key problem in software development at all scales.
  - If dependency is the problem, *duplication* is the symptom.
- **Duplication**:
  - E.g.: logic (same conditional expression in multiple places in the code).
- Eliminating duplication in programs eliminates dependency -> higher chances to pass next test. E.g. Using:
  - Objects (excellent to abstract away the duplication of logic).
  - Symbolic constants

Test Driven Development

- You have defined the behaviour you expect from your “Conversion” functions.
- Now: you're going to write a test suite that puts these functions into stress and makes sure that they behave the way you want them to.
- **YES! You're going to write code that tests code that you haven't written yet.**
- This is called **unit testing**
  - since the set of two conversion functions can be written & tested as a unit, separate from any larger program they may become part of later.

Unit Testing

- Important part of an overall **testing−centric development** or **Test Driven Development (TDD)** strategy.
- Not replacement for higher−level functional or system testing, but important in all phases of development
- If you write unit tests:
    - Write them **early** (preferably before writing the code that they test)
    - Keep them **updated** as code and requirements change.

Benefits of Unit Testing

- **Before** writing code, forces you to detail your requirements in a useful fashion.
- **While** writing code, keeps you from over−coding.
    - When all test cases pass, the function is complete.
- **When refactoring** code, assures you that the new version behaves the same way
- **When maintaining** code, helps you cover your ass when someone comes screaming that your latest change broke their old code ("But all the unit tests passed when I checked it in...")

Benefits of Unit Testing

- **When writing** code in a team, increases confidence that the code you're about to commit isn't going to break other people's code, because you can run their unittests first.
- Seen in **code sprints**:
  - A team breaks up the assignment,
  - Everybody takes the specs for their task,
  - Everybody writes unit tests for it and shares their unit tests with the rest of the team => Nobody goes off too far into developing code that won't play well with others

Unit Testing: Test First Philosophy

- Create test cases before writing code
- Tests must run flawlessly for development to continue
    - “We only write new code when we have a test that does not work” [Jeffries01]

TDD Process

Repeat

- Select functionality to implement
- Create and/or modify system-level tests
- Repeat
  - Developer selects one unit-level module to write and/or modify
  - Developer creates and/or modifies unit-level tests
  - Repeat
    - Developer writes and/or modifies unit-level code
  - Until all unit-level tests pass
- Until all system-level tests pass

Until no more functionality to implement

Unit Testing Python Modules

Test runners examples:

- `unittest`
- `nose or nose2`
- `Pytest`
- `PyUnit`

Important: Choosing the best test runner for your requirements & experience level

Our choice: Unittest Python Module

- Python's framework for unit testing (Included from Python 2.10
- Contains both a testing framework and a test runner.
- Unittest requires you to:
  - Put your tests into classes as methods
  - Use a series of special assertion methods in the unittest.TestCase class instead of the built-in assert statement
- unittest — Unit testing framework Documentation. https://docs.python.org/3/library/unittest.html

Unittest Python Module assert methods

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)`

`.assertIs()`, `.assertIsNone()`, `.assertIn()`, and `.assertIsInstance()` all have opposite methods, named `.assertIsNot()`, and so forth.

https://realpython.com/python-testing/

Unit Testing

- Example of Test Class

```
import roman2
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
```

Unit Testing

- Example of Test Method

```
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'))

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 4000)
```

Unit Testing

- Example of exception definition

```
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass
```

Unit Testing

- Example to run all tests:

Python test_roman.py

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

How to write a test for a method that does not exist?

Strategies for quickly getting to green:

- **Fake It** - return a constant and gradually replace constants with variables until you have the real code
- Obvious Implementation

How to test a program not created yet? A fast stub

A quick dummy function will solve that problem.

```
# roman5.py

def from_roman(s):

    '''convert Roman numeral to integer'''
```

We can define a function with nothing but a docstring. That's legal Python.

Now the test cases will actually fail.

```
$ python3 test.py
```

Unit Testing -Summary-:

- Gives you confidence to do large−scale refactoring (even if you didn't write the original code).
- A powerful concept which, if properly implemented, can both reduce maintenance costs and increase flexibility in any long−term project.
- It is NOT:
  - a panacea: Writing good test cases is hard
    - Keeping them up to date takes discipline (esp. when users scream for critical bug fixes).
  - a replacement for other forms of testing (functional testing, integration testing, user acceptance testing).
- Once you see it work, you wonder How you ever got along without it?

Practical time! You will:

1. Learn TDD through the excellent Dive into Python3 Chapter 9 Unit Testing Tutorial.
   a. Support chapter: Regular Expressions. (If time, also Chapter 10: Refactoring)
   b. Create the methods asked (you can download referred programs from TDD/roman folder)
2. Extra (Excellent tutorial): Real Python https://realpython.com/python-testing/ (first 3 sections, exclude Flask section)
3. Within repo **IN104_NameA_SurnameA_NameB_SurnameB**, create a folder called "TDD" containing your Python programs
4. Create tests with the same philosophy in your last weeks projects
5. Show programs & outputs to your Teaching Assistant (TA)
6. Send the link to your repository to your TA
7. Write tests with the same paradigm in your final project

References

1. Kent Beck. Test-Driven Development by Example.
2. TDD, Software Development Best Practices, Construx.

APPENDIX

E.g. Java example: Apples and Oranges

```
public void testEquality() {
        assertTrue(new Dollar(5).equals(new Dollar(5)));
        assertFalse(new Dollar(5).equals(new Dollar(6)));
        assertTrue(new Franc(5).equals(new Franc(5)));
        assertFalse(new Franc(5).equals(new Franc(6)));
        assertFalse(new Franc(5).equals(new Dollar(5)));
}
```

It fails. Dollars are Francs. Before you Swiss shoppers get all excited, let's try to fix the code. The equality code needs to check that it isn't comparing Dollars and Francs. We can do this right now by comparing the class of the two objects—two Moneys are equal only if their amounts and classes are equal.

Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount && getClass().equals(money.getClass());
}
```