Test Driven Development (TDD) & Python Unit Tests

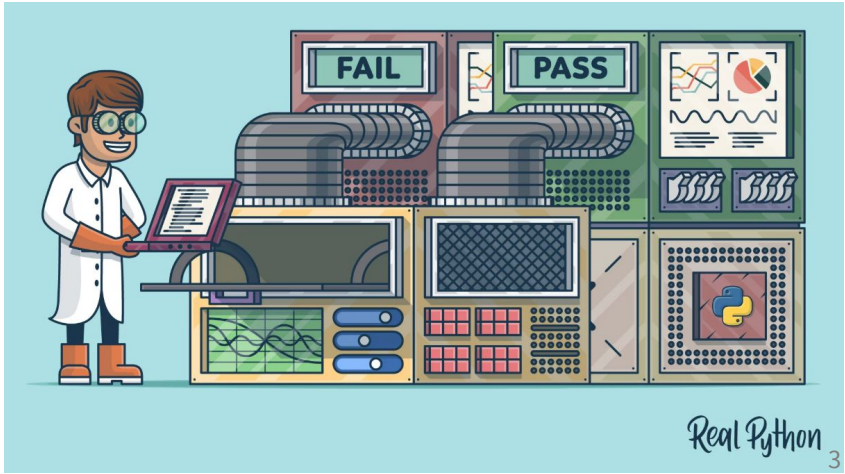
IN104

Natalia Díaz Rodríguez, ENSTA ParisTech

"More than the act of testing, the act of designing tests is one of the best [defect] preventers known … The thought process that must take place to create useful tests can discover and eliminate problems at every stage of development" Boris Beizer

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



Test Driven Development (TDD): Test First Philosophy

- Create test cases before writing code
  - Business representatives write **acceptance** tests to demonstrate that user stories are correctly implemented
  - Programmers continually write **unit** tests which must run flawlessly for development to continue
- *“We only write new code when we have a test that doesn’t work”* [Jeffries01]

Test Driven Development (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

But why TDD?

- One of the biggest problems in software is requirements **ambiguity**
  - A direct result of using natural language specifications
    - e.g.: “The system shall be *fast*”
- A test case is inherently unambiguous
  - => unambiguous “proxies” for requirements

Transitioning to TDD: *Test A Little, Code A Little*

- You don’t need to write all the test cases first
  1. You only have to create **one test** that the current code won’t pass
     a. Write tests for the **parts you are adding or changing**
     b. Write tests for the parts that are **causing you problems**
     c. Gradually, you’ll build up a set of tests
  2. Write code to pass the test
  3. Go to step 1
- You may find the code isn’t designed to make writing tests easy

=> There may be a design issue requiring refactoring or rewriting

- Each **round** of test & code includes refactoring of the previous code

TDD Rationales: Dependency & 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 logic duplication).
  - Symbolic constants

Test Driven Development

- Suppose: You have defined the behaviour you expect from your “**Currency 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 **TDD/ testing−centric development** 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 they test)
  - Keep them **updated** as code and requirements change.

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

Testing from Use Cases

- **Positive** tests
  - Normal course
  - Each alternative course
  - Combinations of alternative courses?
  - Validate the assumptions?
- **Negative** tests
  - Violate each precondition
  - Force each exception

Benefits of Unit Testing

- **Before** writing code, forces you to **detail your requirements** in a useful fashion.
- **While** writing/refactoring code, keeps you from over−coding.
    - 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.
    - When all test cases pass, the function is complete.
- **When maintaining** code, helps you cover your a** 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

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

Benefits of Unit Testing

- Gradually builds an comprehensive **suite** of (hopefully automated) test cases
  - Run that suite each time the code is compiled
  - All tests must pass except the brand new one(s)
- Saves time during integration and system testing
  - Most tests can be run automatically
  - **Integration** errors can be found before **system** test

Unit Testing: Example of Test First Development

We need a method that finds the largest number in an array

```
int largest(int[] list)
```

- Given [7, 8, 9] it should return 9

Unit Testing: Example of Test First Development

Example Test Cases

- [7, 8, 9] => 9
- [8, 9, 7] => 9 Order shouldn't matter
- [9, 7, 8] => 9
- [7, 9, 8, 9] => 9 Multiple occurrences are OK
- [1] => 1 One element is still an array!
- [-9, -8, -7] => -7 Negative numbers are OK

QA-level TDD

Use Case Description Template

Use case # nnn	Use case name here
Actor(s)	Identify which actors can access use case
Description	Give an overall description of intent of use case
Preconditions	Identify what must be true at the start of use case to complete successfully
Postconditions	Identify what must be true on completion of use case to complete successfully
Priority	State how important use case is relative to all of other use cases in the system (note: this could be interpreted as either execution priority or development priority)
Normal course	Describe typical execution scenario, if important
Alternative courses	Identify any nontypical execution scenarios that still constitute successful completion of use case. These are different ways that postconditions could still be satisfied
Exceptions	Identify any execution scenarios that constitute unsuccessful completion of this use case. These are different ways that, despite the preconditions having been satisfied, the postconditions will not have been achieved.
Special requirements	List any other specific (i.e., nonfunctional) requirements that apply to use case
Assumptions	Identify any assumptions behind specification of use case

QA-level TDD

Use Case Description Template: Example

Use case # 66	Make Flight Reservation(s)
Actor(s)	Travel Agent, Traveler, Airline Ticket Agent
Description	Make a reservation on one or more requested flight segments in name of specified traveler.
Preconditions	Each specified flight segment exists. There is room available in fare/class for each flight. All applicable advance-purchase/stay requirements are satisfied.
Postconditions	Each reservation instance has been created. The reservation confirmation has been given to actor. Space available in each fare/class for each flight has been reduced.
Priority	Highest
Normal course	Reservation request is made and completed.
Alternative courses	Actor pays for reservations immediately. Actor preassigns seats. Actor requests special meal. Actor requests extra service like wheelchair or unaccompanied minor service (traveler is under 12 years old).
Exceptions	One or more minimum airport connection times isn't satisfied. Traveler is on the FBI watch list.
Special requirements	Must be completed in under 60 seconds.
Assumptions	This use case only covers making reservations for one person at a time. It also doesn't address related travel services like rental car and hotel reservations.

Testing from Use Cases

TC1 (Positive, normal course)

- Reservation on existing flight(s) with room in fare/class and advance purchase/stay requirements met → reservation created, confirmation provided, availability reduced

TC2 (Positive, alternative course 1)

- Reservation on existing flight(s) with room in fare/class and advance purchase/stay requirements met & pay immediately → …

TC3 (Positive, alternative course 2)

- Reservation on existing flight(s) with room in fare/class and advance purchase/stay requirements met & pre-assign seat(s) → …

TC4 (Positive, alternative course 3)

- Reservation on existing flight(s) with room in fare/class and advance purchase/stay requirements met & special meal→ …

TC5 (Positive, alternative course 4)

- Reservation on existing flight(s) with room in fare/class and advance purchase/stay requirements met & unaccompanied minor→ …

TC6 → 16 (Positive, all other combinations of alternative courses)

- …
- Reservation on existing flight(s) with room in fare/class and advance purchase/stay requirements met & pay now & pre-assign seat(s) & special meal & unaccompanied minor→ reservation created, confirmed, availability--, paid, seat assigned, meal rqst'd, and UM

Testing from Use Cases

TC17 (Negative, violate precondition 1)

- Reservation on non-existing flight(s) → "Flight doesn't exist"

TC18 (Negative, violate precondition 2)

- Reservation on existing flight(s) and advance purchase/stay requirements met but no room in fare/class → "No room in fare/class"

TC19 (Negative, violate precondition 3)

- Reservation on existing flight(s) with room in fare/class but advance purchase/stay requirements not met → "Advance purchase/stay not met"

TC20 (Negative, force exception 1)

- Reservation on existing flights with room in fare/class and advance purchase/stay requirements met but minimum connect time not met → "Minimum connect time not met"

TC21 (Negative, force exception 2)

- Reservation on existing flight(s) with room in fare/class and advance purchase/stay requirements met but traveler on FBI watch list → "Traveler on FBI Watch List"

Unit Testing in Practice

Examples of Python libraries or Test Runners

- `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 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 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

Unit Testing Philosophy:

Be comfortable:

- Calling **unittest.main()** in your **if __name__** clause to run all your test cases at once

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

Example to run all tests:

`Python test_roman.py`

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 write a test for a method that does not exist?

- Make a **list** of the tests we know we need to have working
- **Tell a story** with a snippet of code about how we want to view one operation
- Made the test compile with **stubs**
  - Made it run by committing horrible sins
- Gradually generalize the working code
- Add items to our **to-do list** rather than addressing them all at once

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

Quick dummy functions solve that problem:

```
# roman5.py

def from_roman(s):

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

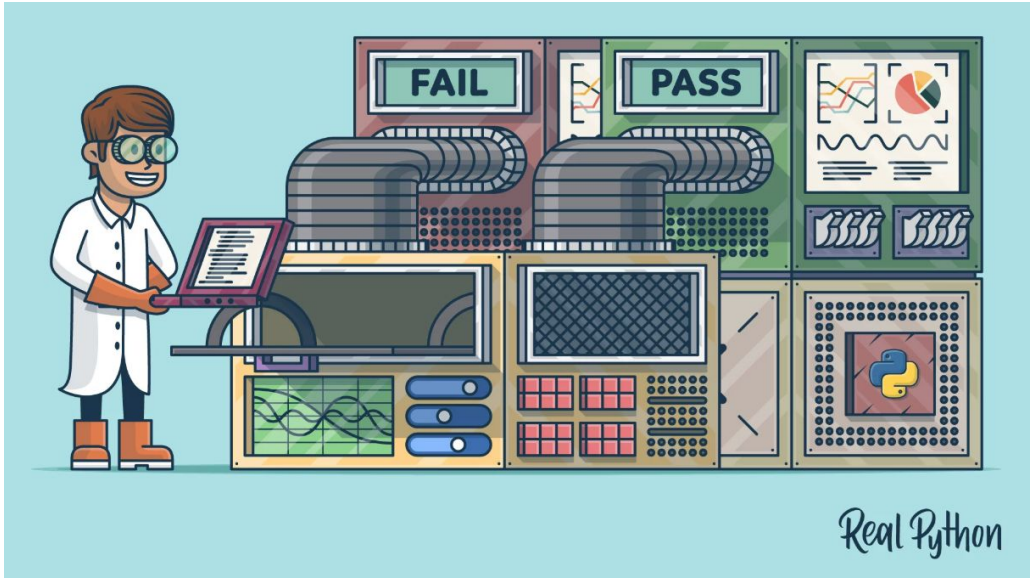
We can define a function with nothing but a docstring. That's legal Python.

Now the test cases will actually fail.

```
$ python3 test.py
```

TDD Cycle: *ReCap*

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 -Summary-:

- Gives you confidence to do large−scale refactoring (even if you didn't write the original code).
- A **powerful** concept
  - If properly implemented, can 1) reduce maintenance costs 2) 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. Follow step by step the excellent Dive into Python3 Chapter 9 Unit Testing TDD Tutorial.
   a. Support chapter: Regular Expressions. (If time, also continue to Chapter 10: Refactoring)
   b. Create the methods asked (you can download referred programs from TDD/roman folder)
2. Follow (first 3 sections, exclude Flask section) of Real Python Testing https://realpython.com/python-testing/
3. Within repo **IN104_NameA_SurnameA_NameB_SurnameB**, create a folder called "TDD" containing your Python programs
4. With the same philosophy, A creates a **unittests_NameA_Surname_A.py** file that instantiates and tests (min 4 different tests) the classes that B created in the previous OOP lecture. B creates **unittests_NameB_Surname_B.py** file that instantiates and tests the classes that A created. Complete/repair the classes of your mate if needed, and commit them.
5. Send the link to your repository to your TA
6. Write tests with the same paradigm in your final project

References

[1] K. Beck. Test-Driven Development by Example.

[2] TDD, Software Development Best Practices, Construx.

[3] M. Pilgrim. Dive into Python3.

APPENDIX

Beyond Python:

E.g. Java example: $ and €

```
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());
}
```