Testing in python tutorial

Prepared by Houndole Lesly & Lucien Zuber
Reviewed by Hunacek Daniel & Widmer Antoine

Date 03.08.2017

Funded by the Horizon 2020 programme
of the European Union

PROJECT INFORMATION

- **Project name** Hotmaps
- **Grant agreement number** 723677

- **Project duration** 2016-2020
- **Project coordinator**

Dr. Lukas Kranzl
TU Wien - Vienna University of Technology
Energy Economics Group – EEG
Gusshausstrasse 25-29/370-3
A-1040 Wien / Vienna, Austria
Phone: +43 1 58801 370351
E-Mail: kranzl@eeg.tuwien.ac.at

Legal notice

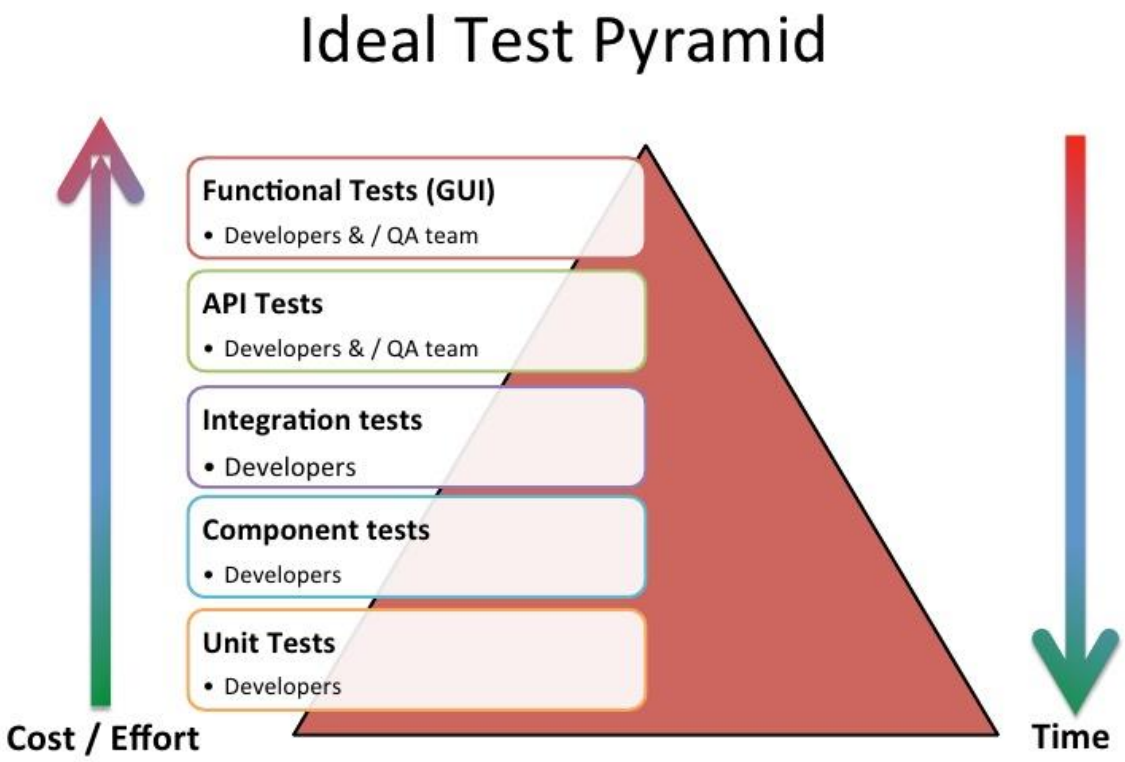
Introduction

The purpose of this tutorial is first to show how to use unit test in **Hotmaps** project. Unit test will answer some important questions in the software development process:

Is this going to work if I insert X?

Does still X return the same results now *that I have implemented Y*?

As you can see unit test is important to maintain, an application, adds new features in an application, avoiding regression also unit test is just the first stage of the test pyramid



In the figure above it is represented the ideal test pyramid. This pyramid can be divided into five different stages of testing:

- Unit tests
- Component tests
- Integration test
- API test
- Functional test

In this document, we will go throughout unit test and integration test; first we will explain the goal of the test then we will describe how does it work and how you can reproduce it. For demonstration, we will use **Python** programming language to describe unit test.

Python is a high-level, interpreted, dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

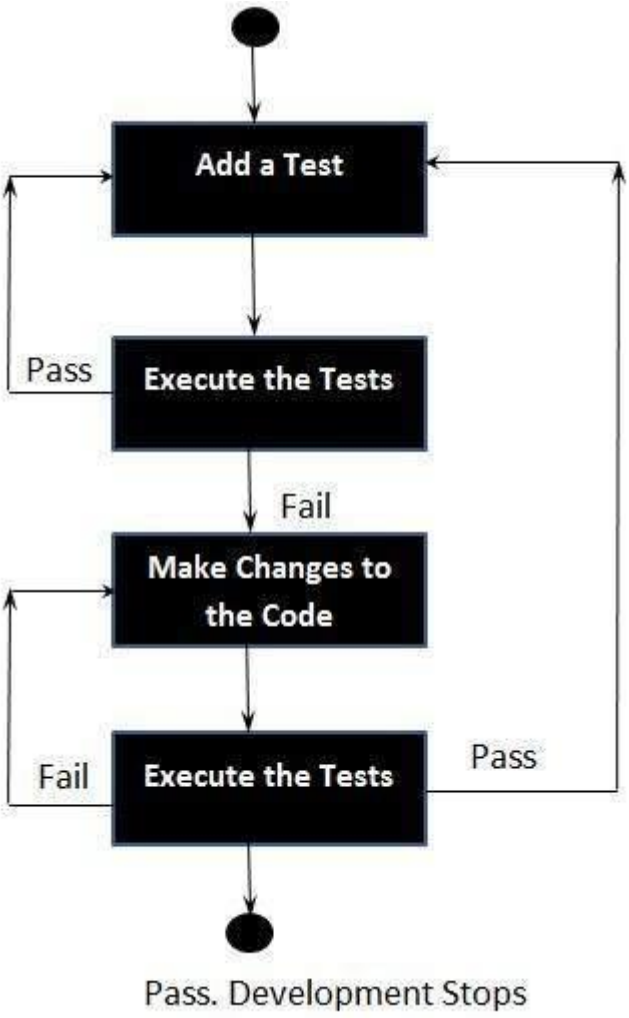
1.Unit test

1.1 Test driven development

Unit test is a component of test-driven development (TDD), a pragmatic methodology that takes a meticulous approach to build a product by means of continual testing and revision. Test-driven development requires that developers first write failing unit tests. Then they write code and refactor the application until the test passes. TDD typically results in an explicit and predictable code base.

The TDD development is of developing a test for each feature of the application. In TDD there are a development process rules to follow:

- Add a test.
- Run all tests and see if the new one fails.
- Write some code.
- Run tests and Refactor code.
- Repeat.



Using TDD gives a lot of benefit like the following:

- Much less debug time.
- Code proven to meet requirements
- Tests become safety.
- Near zero defects
- Shorter development cycles

1.2 Introduction to unit Test

Unit test is a software development process in which features parts of an application, called units, are individually and independently tested for proper operation. Unit test can be done manually but is often automated.

Unit test involves only those characteristics that are vital to the performance of the unit under test. This encourages developers to modify the source code without immediate concerns about how such changes might affect the functioning of other units or the program. Once all the units in a program have been found to be working in the most efficient and error-free manner possible, larger components of the program can be evaluated by means of integration testing.

The great benefit to unit testing is that the earlier a problem is identified, the fewer compound errors occur. A compound error is one that doesn't seem to break anything at first, but eventually conflicts with something down the line and results in a problem.

Let's have a look to unit test in practice.

1.3 Unit test with python

We will use pytest 3.2.0, a framework that improves Python's unittest in order to simplify the tests. Python's unittest, sometimes called **"PyUnit"**, is based on the **XUnit** framework design by Kent Beck and **Erich Gamma**. The same pattern is used in many other languages, including C, Perl, Java, and Smalltalk. **Unittest** supports fixtures, test suites, and a test runner to enable automated testing for your code.

1.3 1 Basic Test Structure

Tests with Pytest are very simple: first, you need to manage your app's requirement in order to import Pytest 3.2.0.

The test itself is composed of two parts: first, you need to execute a function of your program, and then you need to compare the result of your program with the expected results.

```
def get(lst, index, default=None):
    try:
        return lst[index]
    except IndexError:
        return default
```

Here is a simple function, you need to give a list of object to this get, an index and a default message, if the index is null, the default message will be given back.

```
a_list = ('apple', 'vanilla', 'chocolate')

def test_get_element():
    element = get(a_list, 0, 'nothing')
    assert element == 'apple'
```

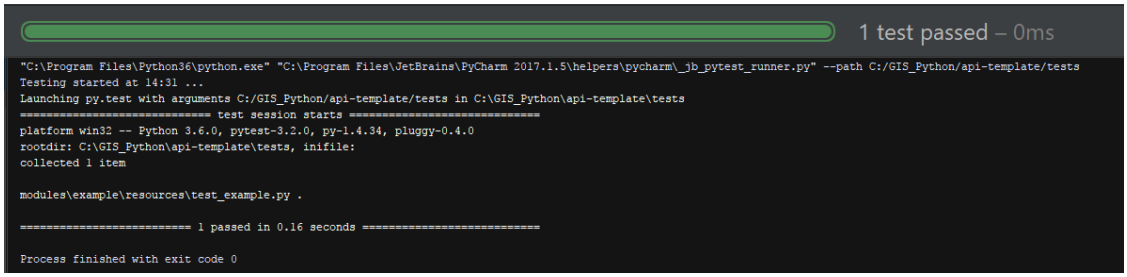
In the code above, you can see a very simple test, it is composed of an expected result (‘apple’) and a function result(element). In this case, we want to see if the first element of our list is apple.

Finally, we use the “assert” function to compare the two results.

1.3.2 Running Tests

To run a test using pytest, you will need to run your test using py.test, with pycharm it can be done by editing your configurations and adding a py.test build.

A successful test will give this output (you can use the “-v” option to obtain a more detailed results)



Test Outcomes

Tests have three possible outcomes:

ok

The test give the awaited response, in our example, the two variables are the same, so the test will pass

```
def test_get_element_missing():
    element = get(a_list, 1000, 'nothing')
    assert element == 'nothing'
```

this code will also pass, because it will display ‘nothing’

FAIL

The test does not pass and raises an AssertionError exception. It means that the output is not what it was supposed to be.

```
def test_get_element_failure():
    element = get(a_list, 1000,'nothing')
    assert element == 'This will not work'
```

this code for instance won't work since it will return 'nothing' and we expect it to return 'This will not work'

ERROR

When the test raise another error, it means that the test has not been correctly configured and you need to fix it (or that the function has crashed to a result)

```
def test_get_element_failure():
    element = get(a_list, 1000
    assert element == 'This will not work'
```

will display

```
============================= test session starts ============================
platform win32 -- Python 3.6.0, pytest-3.2.0, py-1.4.34, pluggy-0.4.0
rootdir: C:\GIS_Python\api-template\tests, inifile:

modules/example/resources/test_example.py:None (modules/example/resources/test_example.py)
C:\Program Files\Python36\lib\site-packages\_pytest\python.py:395: in _importtestmodule
    mod = self.fspath.pyimport(ensuresyspath=importmode)
C:\Program Files\Python36\lib\site-packages\py\_path\local.py:662: in pyimport
    __import__(modname)
E     File "C:\GIS_Python\api-template\tests\modules\example\resources\test_example.py", line 48
E       assert element == 'This will not work'
E            ^
E   SyntaxError: invalid syntax
collected 0 items / 1 errors
```

Since the query is false

1.3.3 Asserting Truth

Note that depending on what you want to obtain, you can play with conditions to alter theh result. For instance this:

```
def test_get_element_failure():
    element = get(a_list, 1000)
    assert element != 'This will not work'
```

will pass since the element won't be equal to 'this will not work'

Test Fixtures

Fixtures are resources needed by a test. For instance, if you are writing several tests for the same class, those tests all need an instance of that class to use for testing. Other test fixtures include database connections and temporary files. **PyTest** includes a special hook to configure and clean up any fixtures needed by your tests. To configure the fixtures, we use @pytest.yield_fixture(), and we need to import pytest

```python
@pytest.yield_fixture()
def a_list():

    print('Before')

    yield ('apple', 'vanilla', 'chocolate')

    print ('after')
```

What is defined here in yield will be called at the beginning of the function, “before” will be called before the yield and “after” will be called at the end of the function.

We also need to mark our fixture as parameter

```python
def test_get_element_missing(a_list):
    element = get(a_list, 1000, 'nothing')
    assert element == 'nothing'
```

and the result:

```
modules\example\resources\test_example.py Before
.after
Before
.after
Before
F
modules\example\resources\test_example.py:54 (test_get_element_failure)
a_list = ('apple', 'vanilla', 'chocolate')

    def test_get_element_failure(a_list):
        element = get(a_list, 1000)
>       assert element == 'This will not work'
E       AssertionError: assert None == 'This will not work'

modules\example\resources\test_example.py:57: AssertionError
after
```

As we can see the code is called after the test.

2 Continuous integrations

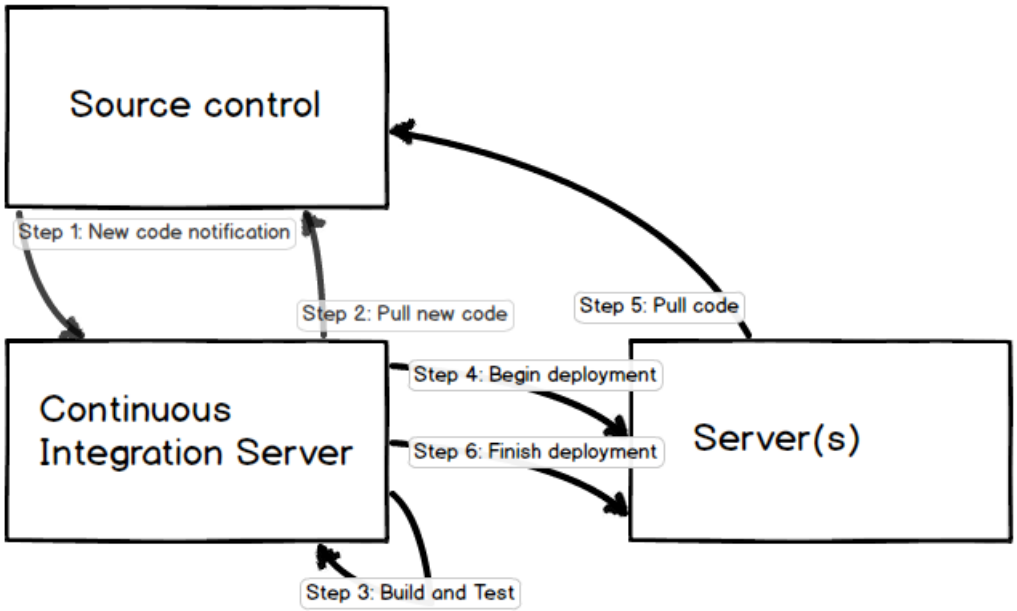
2.1 introduction

Continuous integration automates the building, testing and deploying of applications. Software projects, whether created by a single individual or entire teams, typically use continuous integration as a hub to ensure important steps such as unit testing are automated rather than manual processes.

When continuous integration (CI) is established as a step in a software project's development process it can dramatically reduce deployment times by minimizing steps that require human intervention. The only minor downside to using CI is that it takes some initial time by a developer to set up and then there is some ongoing maintenance if a project is broken into multiple parts, such as going from a monolith architecture to micro - services.

Unit test can be an automated step in the deployment process. Broken deployments can be prevented by running a comprehensive test suite of unit and integration tests when developers check in code to a source code repository. Any bugs accidentally introduced during a check-in that are caught by the test suite are reported and prevent the deployment from proceeding.

CI combined with unit and integration tests check that any code modifications do not break existing tests to ensure the software works as intended.



In the above diagram, when new code is committed to a source repository there is a hook that notifies the continuous integration server that new code needs to be built (the continuous integration server could also pull the source code repository if a notification is not possible).

The continuous integration server pulls the code to build and test it. If all tests pass, the continuous integration server begins the deployment process. The new code is pulled down to the server where the deployment is taking place. Finally, the deployment process is completed via restarting services and related deployment activities.

There are many other ways, a continuous integration server and its deployments can be structured. The above was just one example of a relatively simple setup.

We showed you how to get started with Jenkins. We will show you how to install and configure plugins and run some Python unit tests.

www.hotmaps-project.eu

Contact

Funded by the Horizon 2020 programme
of the European Union