Continuous Integration INRIA

Python Exercises

Vincent Rouvreau - https://sed.saclay.inria.fr *

February 28, 2017

Contents

Python Exercises

1 Preamble

In this exercice, you will learn how to install your `Python` program with packaging tools, test it, measure the tests code coverage, and perform some static code analysis. And finally do this in a single command for multiple `Python` versions and environments.
In the same time, you will use all these tools on `Jenkins` and use visualization plugins for their output on it.

1.1 Tools presentation

For this we will use the following tools :

1. `setuptools` to package your project
2. `unittest` to write tests (standard library)
3. `nose` to run tests and format the results
4. `mock` allows replacing function or objects during tests
5. `pylint` provides static code analysis and style checking
6. `pep8` verifies code compliancy to Python PEP8 style convention
7. `tox` is a virtual environment based test automation tool.

*This practical tutorial has been originally written by Maurice Brémond, Gaëtan Harter and David Parsons from SED INRIA Rhône-Alpes. I would like to thank them for their help and support.

2 Unit testing

2.1 Local setup

Clone the git repository from INRIA forge and create your own branch

First, you will create your personal `git` repository on the INRIA forge as a branch of the main repository for the `TPCISedSaclay` project:

- Go to https://gforge.inria.fr/projects/tpcisedsaclay
- Click on the **CODE SOURCE** or **SCM** tab
- Click on **Request a personal repository**
- Back to the **SCM** tab, look for the command to access your personal repository

WARNING : do not use the anonymous access (containing anon-scm)

Then on a terminal, clone the content of your personal `git` repository. The correct command should look like this:

```
git clone \
  git+ssh://<yourforgelogin>@scm.gforge.inria.fr/gitroot/
  tpcisedsaclay/users/<yourforgelogin>.git
cd <yourforgelogin>/cxx
```

Project file tree

You should have retrieved the following file tree :

```
<yourforgelogin>/python
|_ sphere
|_ tp_ci_sed
|  |_ init .py
|  |_ sphere.py
|  |_ test
|       |_ init .py
|       |_ sphere test.py
|_ raw
   |_ #file used later
```

3 Project presentation

This project aims to compute a sphere volume. As simple as it can be, we want it packaged, tested, and we want to have a nice `build|passing` icon on `github`. Your goal is to create a package, named `tp_ci_sed`, with a `sphere` submodule and a command line interface (or `cli`) script to run it.

3.1 Exercise 1

1. Try the sphere script with a radius as argument.

3.2 Exercise 2

1. Write the `Sphere.volume` method.
   For that, use `math` module and remember volume formula you learned in school: $\frac{4\pi R^3}{3}$ [1].
2. Commit your code with a meaningfull message and push.

3.3 Releasing your code

Now, is your package ready to use by everyone ? Can it be installed by users ?

4 Packaging your code

The first step of an integration process is being able to install the project with its dependencies. In `Python` it is done using a `setup.py` script written with `setuptools` [2].
The script simply calls `setuptools.setup` function with the package description : package directory, name, description, author, license, version, dependencies, supported Python versions, . . .

4.1 Usage

This `setup.py` script is the entry point for installing and releasing your code. It will also be used later as an entry point for tests scripts.

```
python setup.py <command> [opts]
# Install the package
python setup.py install
# Install the package in a way that lets you edit the sources
```

[1] Volume of a 1.5 radius sphere is around 14.137

[2] Default `setup.py` package is `distutils` but it does not manage automatic dependency installation

```
python setup.py develop
# Upload your last revision to \texttt{PyPi} repository
python setup.py upload
```

Setuptools Documentation

https://setuptools.readthedocs.io/en/latest/

4.2 Writing the `setup.py` script

Writing the script is basically just giving the right information in the right format. Usually adapting an existing one is simple enough so it is provided in this exercise. Copy the `first` script from the `raw` folder as `setup.py`.

```
cp raw/first setup.py
cat setup.py
#! /usr/bin/env python
# -*- coding:utf-8 -*5

import os
from setuptools import setup, find_packages
...
```

4.3 Exercise 3

1. Open the `setup.py` script and look at the `setup` function arguments.

Package dependency

Instead of writing in a `README` file explaining which packages are required by your project, those packages can be automatically installed with `setuptools`. Dependencies should be listed using `install_requires` parameter. In the example below, `argparse` dependency will be installed with the package.

```
setup(
    ...,
    install_requires=['argparse'],
)
```

Depending on your needs, you can specify the version with ==, <, != operators.

Note

By default `setuptools` uses `PyPi` to find packages but can be configured to install from many places, like an existing `git` repository or an `http` url [3].

Package version

`Python` package version is usually accessible at `package_name.__version__` should be consistent with the version in `setup.py`. Importing the package in `setup.py` script is a source of import and code coverage errors [4].

To prevent that, package `__init__.py` file can be read and parsed manually to find the version.

Package version can be obtained using :

```
python setup.py --version
```

4.4 Exercise 4

1. Find where the version is stored and how it is retrieved in the `setup.py` script.

5 Semantic versioning

Versioning should help users follow features addition and know when incompatible changes have been introduced. Semantic Versioning gives you simple rules for changing your version numbers.
From http://semver.org/ summary : Given a version number MAJOR.MINOR.PATCH, increment the :

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build meta-data are available as extensions to the MAJOR.MINOR.PATCH format. Later on, remember to increment the version when you change the API. However, as the code is still in `alpha` so you should increment MINOR instead of MAJOR.

[3] http://python-packaging.readthedocs.io/en/latest/dependencies.html
[4] Don't import your package in `setup.py` https://stackoverflow.com/q/11279096/395687

6 Tests

Executing your code in command line and visually verifying the result consists in testing. However some parts of the code may be hard to reach from the `cli`, and are hard to test this way. Also, we want to automate this step and make it repeatable.
So basically, we need to write code to execute our code and verify the output. And to simplify the organization we will use existing tools to write them.
In this section you will write tests using `unittest` and execute them with `nosetests` tool. Install the nose tools dependencies:

```
sudo -H pip install --upgrade nose nosexcover
```

The test `test_volume_0` has already been written as an example. Run it with `nosetests`:

```
nosetests -v
```

The command result:

```
test_volume_0 (tp_ci_sed.test.sphere_test.TestSphere) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.006s
OK
```

Now go in `tp_ci_sed/test/sphere_test.py` to see the implementation.

`unittest framework` [5] is the `Python` implementation of `JUnit`.
It provides a `TestCase` base class to help writing tests. This class implements several assert functions and also runs `setUp` and `tearDown` methods before and after each test if implemented. This helps configuring a test environment.

6.1 Exercise 5

`test_volume_0` is not enough to validate your code.

1. Add another test to validate the volume computation with:
   $radius = 1.5$ where $volume = 14.137166941154069$
2. Commit your code with the new test.

[5] https://docs.python.org/2/library/unittest.html

6.2 Jenkins setup

Log into the project's Jenkins instance

1. Connect to the INRIA Continuous Integration Web portal : https://ci.inria.fr/.
2. Log in and click **Dashboard** in the top menu
3. As you have been added to project **TPCISedSaclay**, click on the **Jenkins** button. You may be prompted to log into `Jenkins`, use the same login/passwd as for the `ci.inria.fr` portal.

Running your first test with Jenkins:

- From our Jenkins dashboard page, click **New Item** in the menu on the left
- Provide a name (`<yourforgelogin>` for instance) for this new item (avoid spaces since it is likely to lead to errors) and select **Freestyle project**.

`Git` configuration :

- In the new item's configuration page (which you will be redirected to after clicking **OK**), choose **Git** as your **Source Code Manager**
- Copy the anonymous URL to your personal repository into the **Repository URL** field:

```
https://scm.gforge.inria.fr/anonscm/git/tpcisedsaclay/users/<yourforgelogin>.git
```

An important step for the continuous integration setup is the build trigger. A simple option is to choose to build periodically : this is suitable for some nightly or weekly tests that may be time consuming and are not meant to be launched after each commit, but this should be avoided for short periods.
In our case, we want the results to be displayed as soon as possible, so we choose to launch the build after a `post-commit` hook [6]:

- Click on **Poll SCM**
- In the **Schedule** field, cut and paste:

```
# Leave empty. We don't poll periodically, but need
# polling enabled to let HTTP trigger work
```

[6] It is explained in ci.inria.fr FAQ documentation

Click on **Save** button.

create the post-commit hook on the INRIA forge server

For this, you can copy the following file:

```
$ ssh <yourforgelogin>@scm.gforge.inria.fr
$ cp /gitroot/tpcisedsaclay/users/vrouvrea.git/hooks/post-receive \
    /gitroot/tpcisedsaclay/users/<yourforgelogin>.git/hooks/
```

And then modify the post-commit hook `post-receive` with your personal repository:

```
#!/bin/sh
wget -q -O - --auth-no-challenge --no-check-certificate \
  http://ci.inria.fr/tpcisedsaclay/git/notifyCommit?url=
  https://scm.gforge.inria.fr/anonscm/git/tpcisedsaclay/users/<yourforgelogin>.git
```

6.3 Exercise 6

Add a new build step using **Virtualenv Builder**, in the job configuration on `Jenkins`.
Note Using **Virtualenv Builder** will put the code execution in a separated `Python` virtualenv environment. This will give a clean `Python` installation and allow installing packages for this build. Now in this field, you inform `Jenkins` how to build and test the project.

```
# Install dependencies
pip install --upgrade nose nosexcover

# Go in source directory
cd python
# Run tests and generate XML test results
nosetests -v --with-xunit
```

Then add Jenkins test result report and build:

1. Click on Post build Actions and select Publish JUnit test result report]
2. For Test report XMLs put **/*tests.xml to scan for all files ending with 'tests.xml'.

3. Then save the project and verify the configuration with a click on Build Now in the menu on the left.
4. In the Build History on the left, click on the last build (hopefully #1), then select Console Output. You can also check the Test Result.

7 Refactoring

Now that your code is tested, you can safely start refactoring it. In our formula, the $\frac{4\pi}{3}$ part can be computed only once instead of at each `volume` call.

7.1 Exercise 7

1. Move it in a class variable and use it in the `volume` method.
2. Run the tests again. It should fail :

```
nosetests -v
```

The command result:

```
test_volume_0 (tp_ci_sed.test.sphere_test.TestSphere) ... ok
test_volume_1_5 (tp_ci_sed.test.sphere_test.TestSphere) ... FAIL

======================================================================
FAIL: test_volume_1_5 (tp_ci_sed.test.sphere_test.TestSphere)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../tp_ci_sed/test/sphere_test.py", line 19, in test_volume_1_5
    self.assertEqual(14.137166941154069, vol)
AssertionError: 14.137166941154069 != 14.137166941154067

----------------------------------------------------------------------
Ran 2 tests in 0.006s

FAILED (failures=1)
```

Note If it does not fail, ask for help, magic numbers technique failed. However, let's assume it is successful and commit your modifications.

```
git commit -m "Precompute 4pi/3"
git push
```

3. The test should now fail on `Jenkins`. See the Console output.

Even if you didn't test it on your computer before committing, at least it is tested somewhere and the error can be detected.

Version Have you thought about increasing the package version?

7.2 Re-factoring error

Why does the test fail? Because of floating point arithmetic rounding errors. Our response to this issue strongly depends on what kind of application we are working on:

1. In some cases, we want to be aware that something has changed, even when the change is the tiniest. In that case, the test we already have is just what we want.
2. In other cases, we need not worry about such a small difference and hence do not want to be bothered by tests complaining.

Here, the code result is correct, so it should pass.

7.3 Exercise 8

1. Find a way to modify the test so small rounding differences do not cause errors
2. Commit, push and verify `Jenkins` result

Hint : See the list of `unittest` assert functions:
https://docs.python.org/2/library/unittest.html#unittest.FunctionTestCase

8 Code coverage

When running tests, the information of what part of the code has been tested is also meaningful. It is often referred to as "code coverage".
With `Python` it can be generated with the `coverage` package, but also directly with `nosetests` and `nose-xcover`.
To get the coverage output, run:

```
nosetests -v --with-xcoverage --cover-erase --cover-inclusive \
--cover-branches --cover-html --cover-package tp_ci_sed
```

```
test_volume_0 (tp_ci_sed.test.sphere_test.TestSphere) ... ok
test_volume_1_5 (tp_ci_sed.test.sphere_test.TestSphere) ... ok
Name                 Stmts   Miss Branch BrPart   Cover   Missing
----------------------------------------------------------------
tp_ci_sed.py             1      0      0      0    100%
tp_ci_sed/sphere.py     19      6      2      1     67%   19, 28-32, 36, 35->36
----------------------------------------------------------------
TOTAL                   20      6      2      1     68%
----------------------------------------------------------------------
Ran 2 tests in 0.012s

OK
```

You can read the coverage output directly in the `nosetests` output. But a human readable html report and an xml report for `Jenkins` are also generated.

Look at the html report using a Web browser :

```
firefox cover/index.html &
```

9 `setup.py` sub-commands

As running `nosetests` with all the options is quite a pain, you will use `setuptools` to configure the `nosetests` command.
Calling `nosetests` can be done using :

```
python setup.py nosetests
```

```
...

reading manifest file 'tp_ci_sed.egg-info/SOURCES.txt'
writing manifest file 'tp_ci_sed.egg-info/SOURCES.txt'
...
----------------------------------------------------------------------

Ran 2 tests in 0.006s

OK
```

Configuring `setup.py` commands requires adding options in a `setup.cfg` file. Copy the `second` script from the `raw` folder as `setup.cfg`.

```
cp raw/second setup.cfg
cat setup.cfg
```

The first part is `nosetests` configuration. It allows coverage and test XML results and also searching for all type of tests (unittest, doctest).
Re-run the command :

```
python setup.py nosetests
```

```
...

----------------------------------------------------------------------
XML: /home/harter/work/tpcisedsaclay/python/nosetests.xml
Name                 Stmts   Miss Branch BrPart   Cover   Missing
----------------------------------------------------------------------
tp_ci_sed.py             1      0      0      0    100%
tp_ci_sed/sphere.py     19      6      2      1     67%   19, 28-32, 36, 35->36
----------------------------------------------------------------------
TOTAL                   20      6      2      1     68%
----------------------------------------------------------------------
Ran 2 tests in 0.012s

OK
```

Now running the tests with `python setup.py nosetests` also generates the coverage report.

9.1 Exercise 9

Modify your item on `Jenkins` to take into account the code coverage.

1. First change the `nosetests -v --with-xunit` build action with `python setup.py nosetests`
2. Add **Post build Action - Publish Cobertura Coverage Report**
3. Configure Cobertura xml report pattern with: `**/coverage.xml`

Workaround As source files are not located at root directory, `Cobertura` plugin fails to find them correctly. A solution is to create a symbolic link to the source directory.

1. In the build step, put at the beginning:

```
# Hack to help Cobertura find source files
ln -nfs python/tp_ci_sed tp_ci_sed
```

2. Commit your code and push
3. Manually re-run a build to get Cobertura plugin graph (it needs more than one build)
4. You can click on the report to see per-file coverage info.

10 Code static analysis and style

`Python` is a dynamic language where many things are evaluated at run-time. So it is different from compiled languages. It means that if you do typos, errors may only occur when running the program. Using a static analysis program can detect errors without executing your program.
Another way to avoid typos and increase readability, is using coding standards for formatting your code. It make all your code look the same and it eases comprehension. For `Python` there is even an official coding style guide defined in the Python Enhancement Proposal 8 (`PEP8`[7]).
In this section you will use two tools to perform static analysis and code style checking : `Pylint` and `PEP8`.
They will be run using the `setup.py` script and so their configuration is set in `setup.cfg`.

1. Install dependencies. Note the setuptools-XXXX which provides setup.py wrapper.

```
sudo -H pip install --upgrade pylint pep8 setuptools-lint setuptools-pep8
```

2. Run the two commands :

```
python setup.py lint
python setup.py pep8
```

You should get a lot of errors and warnings.

[7] https://www.python.org/dev/peps/pep-0008/

10.1 Exercise 10 - Pylint/Pep8 on Jenkins

You will now run `pylint` and `pep8` on `Jenkins`. However the problem with `pylint` and `pep8` commands are that they return a non zero value when there is a warning.
To prevent that, the return value will be hidden, because, in this case, it should not be fatal.

1. Add `pylint` and `pep8` in `Jenkins` VirtualenvBuilder build step as:

   ```
   python setup.py lint | true
   python setup.py pep8 | true
   ```

   The `' | true'` will silent the return value as they are not fatal steps. (It's a choice, I have projects where `pylint` errors are considered failures).

2. To get the reports use **Post build Action** named `Scan for compiler warnings` and in `Scan console log`, add `pylint` and `pep8`.

3. Re-run the build manually.

10.2 Exercise 11

1. Fix some of the errors, but do not spend too much time. However, putting an empty docstring to get rid of the message is evil, it is your code documentation !

2. Commit and see the output on `Jenkins`.

Note The tools may return some false-positive. In this case it is possible to disable the error messages. Refer to the documentation when needed : hrefhttp://docs.pylint.org/message-controlhttp://docs.pylint.org/message-control

11 All tests in one command

Now that we sum up what we have, testing all your application with the previous tools is simply running the three following commands:

```
python setup.py nosetests
python setup.py lint
python setup.py pep8
```

You can write that in your documentation and let other users to do it. Is it enough?
Do you remember to tell to install the tests dependencies? And if you want to add another test tool? You must inform all developers to run the new command.
And if you want to test with `Python3` ?
Adding a bash script to do it would work but as we are working with `Python`, let's use the right tool.

tox[8] is a generic `virtualenv` based automation tool. It lets you test that your package can be installed with different interpreters and runs tests in each environment.

1. Install `tox` dependency.

```
sudo -H pip install --upgrade tox
```

2. Copy the `tests utils` from `raw` directory:

```
cp -r raw/tests_utils .
ls tests_utils/
```

```
coverage-python-3.2.txt pylint-python-2.6_3.2.txt test-requirements.txt
```

3. Look at these files. All tests dependencies are described in the `test-requirements.txt` file. The two other files are specific versions for `Python` 2.6 and 3.2.

4. Copy the tox.ini file from raw/third file.

```
cp -r raw/third tox.ini
```

The file content is the following : Listing 1: tox.ini

```
#
# This file is a part of TPCI_SEDRA project
# Copyright (c) INRIA 2015
#
# Contributor (s) : Gaetan Harter
# Contacts : gaetan.harter@inria.fr
#
```

[8] https://tox.readthedocs.org/en/latest/

```
# This project is headed by SED Rhone-Alpes service at INRIA.
#
[tox]
envlist = py27
# envlist = py26,py27,py32,py33,py34

[testenv]
deps=
-rtests_utils/test-requirements.txt
py26,py32: -rtests_utils/pylint-python-2.6_3.2.txt
py32: -rtests_utils/coverage-python-3.2.txt
commands=
python setup.py nosetests
-python setup.py lint
-python setup.py pep8
```

The different part of the file are:

(a) `envlist` : is the list of `Python` version you want to run on. Here we only consider `Python` 2.7 for the moment.

(b) `deps` : the test dependencies. With specific packages versions for `Python` 2.6 and 3.2.

(c) `commands` : the list of test commands. The front dash - means "don't fail" on execution error.

From now, executing your tests simply means running `tox` and the only required dependency installed is `tox` itself.

5. Commit and push.

11.1 Exercise 12 - Tox on Jenkins

1. Update Jenkins main build step. You can replace all dependencies installation with only tox and replace the setup.py execution with tox.

```
# Hack to help \texttt{Cobertura} find source files
ln -nfs python/tp_ci_sed tp_ci_sed
pip install --quiet --upgrade tox
cd python
tox
```

2. Re-run the build.

For me, tox is the most important step of this session.
It really solves common problems and simplify testing by doing test dependency

management, document in a script the test procedure(s) and that it also handles many tests environments in one command for the user.

Other versions If everything succeeds, and you can test using `Python` 3 (`Python` 3 is installed in the virtual machine related to `ci.inria.fr/tpcisedsaclay`).

1. Use `envlist = py27,py32` (for Python3.2)
2. Re-run `tox`.

Tests should fail for `Python` 3, at least for the print statement. However, you see how easy it is to test your code on different `Python` versions. Making a code working for `Python` 2 and 3 requires some time, but at least it is easy to test.

12 Advanced tests writing : mocking

mock http://www.voidspace.org.uk/python/mock/index.html
When testing, it is useful to replace some part of your application under test to prevent its side effect, like connection to an external service, or make an `open` call raise an exception.
It is possible to fill your whole hard-drive to get an `IOError: No space left on device` but it might be better to just modify `open` to simulate it to test your code.
It is also useful to be able to know how an object has been used or a function been called, the same way as testing what has been printed by your program.
In this section you will see `mock` to test the `main` function and verify the printed output.

Testing main In this case, testing the `main` function ie calling the sphere script from command line, means:

1. Set the command line arguments to some values : `sys.argv`
2. Verify the printed text : in files `sys.stdout` and `sys.stderr`

Install `mock` dependency :

1. Install the dependency on your system. It will be installed by `tox` in the test environment.

```
sudo -H pip install --upgrade mock
```

`mock` class provides mainly two important things:

1. `Mock` class which is an object whose behavior can be configured and that records all action on it.

2. `patch` function that dynamically replaces one function/method/object with a `Mock` object.

Remark Explaining `mock` and applying is too long for this course. So in this section you have to copy paste the code, try it and read afterwards to understand the example. See online documentations when trying it yourself.

1. Copy the fourth file as tp_ci_sed/test/sphere_test_main.py
2. Re-run tests.

```
cp -r raw/fourth tp_ci_sed/test/sphere_test_main.py
tox
```

You can see that we now have 100% coverage.
Now open the sphere_test_main.py in a text editor and check this:

Main function In order to test the main function, the program arguments `sys.argv` should be replaced with different values. Here I am patching it using a `context_manager` (with keyword). It automatically applies and cleans up the patch.

Mocking outputs Two methods are presented:
The first one is mocking `sys.stdout` with a `StringIO` object and get the written string. Here it is also done using a context manager.
Another, is mocking the file object and verify `write` call arguments. It is presented here for `sys.stderr` by patching in `setUp` and cleaning it with `patch.stopall()` in `tearDown`.
Both method are valid, it is just for presenting alternatives.

Hints Some advises on patch :

1. When "patching" always verify that the mock has been called by your program, there are so many cases where you can just fail to patch correctly.
2. When patching a class, you should use `MockClass.return_value` to get the instance `mock`.
3. The patching path can depend on how the module is imported, see: http://www.voidspace.org.uk/python/mock/patch.html#where-to-patch. Read it, try it, and understand this prior testing your application. Basically, when patching an object imported using `import XXXX` it requires absolute patch path. And when an object is imported using `from XXX import yyyyy`, it requires patching from the importing module.

For every other problems, look at the documentation.

12.1 Code review by ChuckNorris

1. Add `ChuckNorris` post build action. It will make your code famous and better.

It displays Chuck Norris facts and a picture of Chuck adapted to your build result (seems like the picture is not displayed but why ?)

12.2 Build Passing icon

There is a plugin for that. It is called `Embeddable Build Status`, however it requires the project to have at least "ViewStatus" to allow everyone to see it. And it is not configured in this project.
But you will set it on your own `Jenkins` for your project !
Always verify on a non-authenticated browser.