simple-sample

Release 0.0.4

Jun 13, 2020

Contents:

This package contains a simple sample of a Python package prototype.

CHAPTER 1

Python prototype

This package contains a simple sample of a Python package prototype. It is part of the educational repositories to learn how to write stardard code and common uses of the TDD.

See the documentation and how to do it on readthedocs. And see the development of this code step by step

- with see-git-steps
- on readthedocs / step by step

1.1 Installation

The package is self-consistent. So you can download the package by github:

```
$ git clone https://github.com/bilardi/python-prototype
```

Or you can install by python3-pip:

```
$ pip3 install simple_sample
```

1.2 Usage

Read the unit tests in tests/testMyClass.py file to use it. This is a best practice. You can read also the documentation by command line,

```
$ python3
>>> from simple_sample.myClass import MyClass
>>> print(MyClass.__doc__)
>>> help(MyClass)
>>> quit()
```

If you want to see the local documentation, that you have downloaded by github, you can use the same steps but before you must to change the directory

```
$ cd python-prototype
```

1.3 Development

It is common use to test the code step by step and unittest module is a good beginning for unit test and functional test.

Test with unittest module

```
$ cd python-prototype
$ python3 -m unittest discover -v
```

1.4 Change Log

See CHANGELOG.md for details.

1.5 License

This package is released under the MIT license. See LICENSE for details.

CHAPTER 2

How to use

In this section you can find some tips & tricks for learning to use any code.

2.1 Unit tests

When you change something on your code, you can run one unit test about that class changed

```
$ cd python-prototype
$ python3 -m unittest -v tests/testMyClass.py
```

And when you are ready for the commit, you can use a command for running all unit tests

```
$ cd python-prototype
$ python3 -m unittest discover -v
```

But for learning how to use an class in your code, you need to read its unit test file. You can find the import class, the initialization class, and the main public methods.

2.2 Documentation

Another approach is to read the documentation in the class file. When you install a package, you can also read its documentation by shell

```
$ python3
>>> import simple_sample
>>> print(simple_sample.__doc__) # description like overview of the package
>>> help(simple_sample) # description of the package contents, and if there are,␣
↪classes and functions
>>> quit()
```

In the description, you can find an example for a command that you can use for each package element

```
$ python3
>>> import simple_sample.myClass
>>> print(simple_sample.myClass.__doc__) # description like overview of the element
>>> help(simple_sample.myClass) # description of the element contents: classes and␣
↪functions
>>> quit()
```

If there are more classes in an package element, you can import a specific class where to read all methods

```
$ python3
>>> from simple_sample.myClass import MyClass
>>> print(MyClass.__doc__) # description like overview of the class
>>> help(MyClass) # description of the class contents: methods, and if there are,␣
↪functions
>>> quit()
```

2.3 MyClass

The precedent approches are the best practice for learning something about a specific package.

Sometimes, the package is so complex, that it is also necessary a "Quick start" where a developer can learn the main classes or methods to start from

```
$ python3
>>> from simple_sample.myClass import MyClass
>>> mc = MyClass() # initialization with or without boolean argument
>>> mc.foo(True); # returns the reverse value
>>> mc.bar(); # returns the boolean initialized
>>> mc.foobar(); # returns the reverse value of the boolean initialized
>>> quit()
```

CHAPTER 3

How to make

In this section you can find how to generate or publish that helps you with managing your code.

3.1 Virtual environment

When you have to install specific requirements.txt, you can use a virtual environment: this method is important to test different packages versions without to install them on your local.

The main commands are

```
$ cd python-prototype
$ python3 -m venv .env # create virtual environment
$ source .env/bin/activate # enter in the virtual environment
$ pip install --upgrade -r requirements.txt # install your dependences
$ deactivate # exit when you will have finished the job
$ rm -rf .env # remove the virtual environment it is a best practice
```

3.2 Documentation

The Python language is permissive, but if you use a basic documentation like Python Enhancement Proposals 8 (PEP-8) and unit / functional test, everyone can easily read your code. There are many style to write Python documentation.

If you load your package on GitHub, GitLab, or BitBucket, you can also use sphinx for creating docs folder like in this package. It can help you to organize concepts in an unique place without duplicates: the README.rst is the homepage on Github and Pypi repositories and it is also one of these pages (see overview.rst).

The main commands for building documentation are

```
$ pip3 install sphinx
$ pip3 install sphinx_rtd_theme # it is necessary only if you want the theme sphinx_
↪rtd_theme
```

(continues on next page)

(continued from previous page)

```
$ cd python-prototype/docs/
$ sphinx-quickstart
$ make html
```

And you can open build/html/index.html in your web browser to see your docs.

Instead, for uploading documentation on readthedocs, you have to follow this guide.

3.3 TDD

Before write code, it is important to verbalize the concepts by documentation and to create Test Driven Development (TDD) for your code. Then, it is important to use unit test for finding the issues and before to update change log file and package version.

See the development of this code step by step on readthedocs / step by step for learning how to make a unit test.

3.4 Packaging

The tutorial for packaging your projects is standard. And then your package is public on PyPI.

The main commands for testing are

```
$ rm -rf build dist *.egg-info
$ python3 setup.py sdist bdist_wheel # create source archive
$ python3 -m twine upload --repository testpypi dist/* # upload source archive on␣
↪testpypi
$ python3 -m pip install --index-url https://test.pypi.org/simple/ --no-deps simple-
↪sample-bilardi # install package from testpypi
```

The main commands for the production environment are

```
$ rm -rf build dist *.egg-info
$ python3 setup.py sdist bdist_wheel # create source archive
$ python3 -m twine upload dist/* # upload source archive on pypi
$ python3 -m pip install simple-sample # install package from pypi
$ pip3 install simple-sample # slim command for installing the package
```

CHAPTER 4

Step by step

This page is for learning all steps that they are necessary for writing a simple package like simple-sample.

4.1 see-git-steps

The package see-git-steps has been written for reading piece by piece the commits of a git repository.

If you want to use it, you have to install it following its README.md.

4.2 Getting started

The goal of the package simple-sample is to create a Python package prototype. So you can use this simple package for downloading a base for your package.

4.2.1 Step 1

The first step is to add all the outline files for your package

```
$ cd python-prototype
$ see-git-steps
6248c90ca6fb91758524addf69a4a179c06baf3d step 1 - add the outline files
.gitignore
CHANGELOG.md
LICENSE
MANIFEST.in
Makefile
README.rst
setup.py
```

They are important files and below you can find a link of a guide for each file

- .gitignore, to ignore specific files when you have to commit
- CHANGELOG.md, the best practise for reading the minor o major change of your code
- LICENSE, the best practise for defining your policy for the public repository
- MANIFEST.in, documentation included into your package
- Makefile, it is not necessary but it is a comfortable way to remember procedures
- README.rst, documentation visible on your repository homepage
- setup.py, it is a best practise for creating or installing your package

The files that you have to customize are

- **.gitignore**, for example, if you use an ide with specific files extension
- **LICENSE**, with year and your name
- **Makefile**, with your PACKAGE_NAME and YOUR_USERNAME of PyPI
- **README.rst**, with your quick start documentation
- **setup.py**, where you find some variables: name, author, author_email, description and urls

When you have modified, you can commit your first change

```
$ cd python-prototype
$ git init # for initializing the repository
$ git add .gitignore CHANGELOG.md LICENSE MANIFEST.in Makefile README.rst setup.py
$ git commit -m "step 1 - add the outline files"
```

4.2.2 Step 2

The second step is to add the first files that you need for creating your package

```
$ cd python-prototype
$ see-git-steps
57e2d766fbdea1d6a3862676f6f3a28e5ab5746c step 2 - add the empty package version
setup.py
simple_sample/__init__.py
tests/__init__.py
```

The files __init__.py are the base of a regular package.

It can be empty, like **tests/__init__.py**, and it will say anyway that folder is a package.

It can contain some code, like simple_sample/__init__.py, and it is sufficient for importing it in setup.py for recovering author and version of the package.

See the changes of **setup.py** by GitHub or by command line with see-git-steps

```
$ cd python-prototype
$ see-git-steps -c 57e2d766fbdea1d6a3862676f6f3a28e5ab5746c -v
```

When you have modified **setup.py** and added the new files, you can commit your changes

```
$ cd python-prototype
$ git add setup.py tests/__init__.py simple_sample/__init__.py
$ git commit -m "step 2 - add the empty package version"
```

4.2.3 Step 3

Before write code, it is important to verbalize the concepts by documentation: so the documentation is important to learn a package as to plan how to write the code.

You can write your documentation as you want: you can create docs folder like in this package, by sphinx.

When you have created your documentation, you can add the new folder and you can commit your changes

```
$ cd python-prototype
$ git add docs
$ git commit -m "step 3 - add documentation by sphinx"
```

When a commit completes one feature or a set of fixies, you can tag that commit as a release. The standard behaviour is to add changes in a CHANGELOG file: see the changes of **CHANGELOG.md** by GitHub or by command line with see-git-steps

```
$ cd python-prototype
$ see-git-steps -c 20b91ae691f29c96059dc3d3b355ab7c91eb9928 -v | head -n 21
```

So you can add CHANGELOG.md on your last commit, or you can create one commit for changelog, and then you can add the tag.

```
$ cd python-prototype
$ git add CHANGELOG.md
$ git commit --amend # add file on your last commit
$ git tag v0.0.1 -m "Empty package and documentation by sphinx" # create a tag with␣
↪that version name
$ git tag -n # show the tag list with description
$ git push origin --tags # load the tag on repository
```

4.2.4 Step 4

Before write code, it is important to verbalize the methods by create Test Driven Development (TDD) for your code. Then, it is important to use unit test for finding the issues and before to update change log file and package version.

In Python, a standard TDD is offered by unittest module: see the unit tests of MyClassInterface by GitHub or by command line with see-git-steps

```
$ cd python-prototype
$ see-git-steps -c b31157739997841621968440f970778059a41946 -v
```

In Python, the interface is not necessary, but this is a simple sample with a bit of everything. The MyClassInterface will have only 2 methods not defined in two different ways: one using **pass** and one using **raise**.

When you have created **tests/testMyClassInterface.py** and added the new file, you can commit your changes

```
$ cd python-prototype
$ git add tests/testMyClassInterface.py
$ git commit -m "step 4 - add the unit test for MyClassInterface"
```

4.2.5 Step 5

Now you can write your first class: see MyClassInterface by GitHub or by command line with see-git-steps

```
$ cd python-prototype
$ see-git-steps -c 31b35d60e49878aa01fd8d7d6c47200d8696523b -v
```

When you have created **simple_sample/myClassInterface.py**, you can run the unit tests of MyClassInterface

```
$ cd python-prototype
$ python3 -m unittest discover -v
test_my_class_interface_can_be_created (tests.testMyClassInterface.
↪TestMyClassInterface) ... ok
test_my_class_interface_gets_bar_value (tests.testMyClassInterface.
↪TestMyClassInterface) ... ok
test_my_class_interface_gets_qux_value (tests.testMyClassInterface.
↪TestMyClassInterface) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

It is a best practise to run unit tests after a change and before a commit. If the result is like the example (all tests are OK), you can add the new file and you can commit your changes

```
$ cd python-prototype
$ git add simple_sample/myClassInterface.py
$ git commit -m "step 5 - add MyClassInterface and unit tests works properly"
```

4.2.6 Step 6

If you need to use a framework, abstract class is a good method. The peculiarity of this is that it cannot be imported. So the unit tests are simple: see the unit tests of MyClassAbstract by GitHub or by command line with see-git-steps

```
$ cd python-prototype
$ see-git-steps -c fcec3fca61c7860a63969c6b6e56d951ab187489 -v
```

When you have created **tests/testMyClassAbstract.py** and added the new file, you can commit your changes

```
$ cd python-prototype
$ git add tests/testMyClassAbstract.py
$ git commit -m "step 6 - add the unit test for MyClassAbstract"
```

4.2.7 Step 7

Now you can write your second class: see MyClassAbstract by GitHub or by command line with see-git-steps

```
$ cd python-prototype
$ see-git-steps -c ddb120bd0c14528b0c8b0caf223b387588725c50 -v
```

When you have created **simple_sample/myClassAbstract.py**, you can run all unit tests like the command of step 5, or you can run only the unit tests of MyClassAbstract

```
$ cd python-prototype
$ python3 -m unittest -v tests/testMyClassAbstract.py
test_my_class_abstract_can_be_created (tests.testMyClassAbstract.TestMyClassAbstract)␣
↪... ok
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

If the test is OK, you can add the new file and you can commit your changes

```
$ cd python-prototype
$ git add simple_sample/myClassAbstract.py
$ git commit -m "step 7 - add MyClassAbstract and unit tests works properly"
```

4.2.8 Step 8

Now you can write the unit tests for a class can extend the interface class and the abstract class: note that it has been also added a unit test for the public method of the abstract class. See the unit tests of MyClass by GitHub or by command line with see-git-steps

```
$ cd python-prototype
$ see-git-steps -c d78aacd6f3ef99d119b8ceb45d2378b1344d5f27 -v
```

When you have created **tests/testMyClass.py** and added the new file, you can commit your changes

```
$ cd python-prototype
$ git add tests/testMyClass.py
$ git commit -m "step 8 - add the unit test for MyClass"
```

4.2.9 Step 9

After verbalizing all MyClass methods by the unit tests, you are ready to write the methods: see MyClass by GitHub or by command line with see-git-steps

```
$ cd python-prototype
$ see-git-steps -c 84ce869d3ea3c5737d99b7b596be3fe4b3d826a4 -v
```

When you have created **simple_sample/myClass.py**, you can run all unit tests

```
$ cd python-prototype
$ python3 -m unittest discover -v
test_my_class_can_be_created (tests.testMyClass.TestMyClass) ... ok
test_my_class_gets_bar_value (tests.testMyClass.TestMyClass) ... ok
test_my_class_gets_baz_value (tests.testMyClass.TestMyClass) ... ok
test_my_class_gets_foo_value (tests.testMyClass.TestMyClass) ... ok
test_my_class_gets_fooquux_value (tests.testMyClass.TestMyClass) ... ok
test_my_class_gets_qux_value (tests.testMyClass.TestMyClass) ... ok
test_my_class_abstract_can_be_created (tests.testMyClassAbstract.TestMyClassAbstract)␣
↪... ok
test_my_class_interface_can_be_created (tests.testMyClassInterface.
↪TestMyClassInterface) ... ok
test_my_class_interface_gets_bar_value (tests.testMyClassInterface.
↪TestMyClassInterface) ... ok
test_my_class_interface_gets_qux_value (tests.testMyClassInterface.
↪TestMyClassInterface) ... ok
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
Ran 10 tests in 0.001s

OK
```

If all test is OK, you can add the new file and you can commit your changes

```
$ cd python-prototype
$ git add simple_sample/myClass.py
$ git commit -m "step 9 - add MyClass and unit tests works properly"
```

4.2.10 Step 10

You are completed the package, so you can tag that commit as a release. This step could be run every time you complete a class with its unit test. The files that you have to update are **CHANGELOG.md**, **docs/source/conf.py** and **simple_sample/__init__.py**, because they contain version number. See the changes by GitHub or by command line with see-git-steps

```
$ cd python-prototype
$ see-git-steps -c 999ce550c0b378011e94a76f654edf851c93ad52 -v
```

So you can add the files updated, you can create a commit dedicated, and then you can add the tag.

```
$ cd python-prototype
$ git add CHANGELOG.md docs/source/conf.py simple_sample/__init__.py
$ git commit -m "step 10 - update changelog and version of the simple-sample package"
$ git push origin master # load the commit on remote repository
$ git tag v0.0.4 -m "The first full version of the simple-sample package" # create a␣
↪tag with that version name
$ git tag -n # show the tag list with description
$ git push origin --tags # load the tag on repository
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search