

# Python Unittest

*Instructors*

**Battista Biggio** and **Luca Didaci**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence  
University of Cagliari, Italy

# Unit Testing Framework

The Python unit testing framework, sometimes referred to as “PyUnit,” is a Python language version of JUnit, by Kent Beck and Erich Gamma. JUnit is, in turn, a Java version of Kent’s Smalltalk testing framework. Each is the de-facto standard unit testing framework for its respective language.

**unittest** supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. The **unittest** module provides classes that make it easy to support these qualities for a set of tests.

# Unit Testing Framework

To achieve this, `unittest` supports some important concepts:

- **test case:** A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs. `unittest` provides a base class, `TestCase`, which may be used to create new test cases.
- **test fixture:** A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.
- **test suite:** A test suite aggregate tests that should be executed together. It is a collection of test cases, test suites, or both.
- **test runner:** A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

# Test Cases

Test cases are supported through the `TestCase` class;

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)
```

# Test Cases and Asserts

A testcase is created by subclassing `unittest.TestCase`.

The test methods are defined with names starting with the string “test”. This naming convention informs the test runner about which methods represent tests.

The `TestCase` class provides several methods to check for and report code failures. Those methods are called **Asserts**. Using those methods, the test runner can accumulate all test results and produce a report.

# Asserts

Some of the assert methods are the following:

Method	Checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x)</code> is True
<code>assertFalse(x)</code>	<code>bool(x)</code> is False
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>

# Asserts

Some of the assert methods are the following:

Method	Checks that
<code>assertAlmostEqual(a, b)</code>	<code>round(a-b, 7) == 0</code>
<code>assertNotAlmostEqual(a, b)</code>	<code>round(a-b, 7) != 0</code>
<code>assertGreater(a, b)</code>	<code>a &gt; b</code>
<code>assertGreaterEqual(a, b)</code>	<code>a &gt;= b</code>
<code>assertLess(a, b)</code>	<code>a &lt; b</code>
<code>assertLessEqual(a, b)</code>	<code>a &lt;= b</code>
<code>assertRegexpMatches(s, r)</code>	<code>r.search(s)</code>
<code>assertNotRegexpMatches(s, r)</code>	<code>not r.search(s)</code>
<code>assertItemsEqual(a, b)</code>	<code>sorted(a) == sorted(b)</code> and works with unhashable objs
<code>assertDictContainsSubset(a, b)</code>	all the key/value pairs in <i>a</i> exist in <i>b</i>

# Test Suite and Test Runner

Test suites are implemented by the **TestSuite** class. This class allows individual tests and test suites to be aggregated; when the suite is executed, all tests added directly to the suite and in “child” test suites are run.

A test runner is an object that provides a single method, **run()**, which accepts a **TestCase** or **TestSuite** object as a parameter, and returns a result object. The class **TestResult** is provided for use as the result object. **unittest** provides the **TextTestRunner** as an example test runner which reports test results on the standard error stream by default. Alternate runners can be implemented for other environments (such as graphical environments) without any need to derive from a specific class.



# A Basic Example of Unittest

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

suite = unittest.TestLoader().loadTestsFromTestCase(TestStringMethods)
unittest.TextTestRunner(verbosity=2).run(suite)
```

# A Basic Example of Unittest

The final block shows the creation of a **TestSuite** that contains all the defined test methods and how those tests can be executed using a **TestRunner**.

The script produces an output that looks like this:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok
```

```
-----
Ran 3 tests in 0.001s
```

OK

# A Basic Example of Unittest

```
suite = unittest.TestLoader().loadTestsFromTestCase(TestStringMethods)
```

This adds to the suite all the methods which are called with a name that starts with the word “test”.

We can add just some of them creating a **TestSuite** class and adding only the methods that we would like to have runned by the test:

```
suite = unittest.TestSuite()  
suite.addTest(TestStringMethods('test_upper'))  
suite.addTest(TestStringMethods('test_isupper'))
```

# Command-Line Interface

The unittest module can be used from the command line to run tests from modules, classes or even individual test methods:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

It is possible to find and run all the test in the sub-directory of a folder:

```
python -m unittest discover -s project_directory -p "test*.py"
```

# Test Fixture

Such test cases can be numerous, and their set-up can be repetitive.

Luckily, the set-up code can be implemented in the `setUp()` which will be automatically call by the framework when we run the test before running each test function. Similarly, we can provide a `tearDown()` method that tidies up after the `runTest()` method has been run.

The operations that are performed during the set-up and cleanup are called test fixture.

# Test Fixture

```
import unittest

def fib(n):
    return 1 if n<=2 else fib(n-1)+fib(n-2)

class TestFib(unittest.TestCase):

    def setUp(self):
        self.n = 10
    def tearDown(self):
        del self.n

    def test_fib_assert_equal(self):
        self.assertEqual(fib(self.n), 55)
    def test_fib_assert_true(self):
        self.assertTrue(fib(self.n) == 55)

suite = unittest.TestLoader().loadTestsFromTestCase(TestFib)
unittest.TextTestRunner(verbosity=2).run(suite)
```

# Skipping Tests

Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as an “expected failure,” a test that is broken and will fail, but shouldn’t be counted as a failure on a **TestResult**.

You can skip an entire test class adding an apposite decorator:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

Or only some specified function as in the following example.

# Skipping Tests

```
import numpy
import unittest
import sys

class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(numpy.__version__ < "1.17",
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        Pass

suite = unittest.TestLoader().loadTestsFromTestCase(MyTestCase)
unittest.TextTestRunner(verbosity=2).run(suite)
```



# Skipping Tests

The output of this test is:

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library version'  
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'  
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'
```

---

Ran 3 tests in 0.005s

OK (skipped=3)

# Expected Failure

It is possible to mark a test as an “expected failure,” a test that is broken and will fail, but shouldn’t be counted as a failure on a **TestResult**. It is sufficient using the apposite decorator.

```
import unittest

class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")

suite = unittest.TestLoader().loadTestsFromTestCase(ExpectedFailureTestCase)
unittest.TextTestRunner(verbosity=2).run(suite)
```

# Git Testing with Continuous Integration

# Continuous Integration (CI)

CI allows automating the execution of tests on different platforms/installations

GitLab and Github offer built-in continuous integration and delivery tools

When a new commit is pushed to the repository, GitLab/GitHub will use their CI runners to execute the test suite against the code in an isolated Docker container

# Gitlab CI



# Setting Up CI/CD on GitLab

To have a working CI you need to:

- add `.gitlab-ci.yml` to the root directory of your repository
- configure a *runner*

The `.gitlab-ci.yml` file tells the GitLab runner what to do. By default, it runs a pipeline with three stages: build, test, and deploy.

The Runner is the process that will trigger the CI pipeline after each commit or push.

A green (if all the test are passed) or red checkmarker will be associated to the given commit. It is then possible to explore the test reports from the GitLab interface

# Creating the `.gitlab-ci.yml` Configuration File

```
image: python:3.5

variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

cache:
  paths:
    - .cache/pip
    - venv/

before_script:
  - python -V                # Print out python version for debugging
  - pip install virtualenv
  - virtualenv venv
  - source venv/bin/activate
  - pip install -r requirements.txt

test:
  only:
    - master
  script:
    - python -m unittest discover -s tests
```

# Configure a Runner

If you use GitLab.com you can use the **Shared Runners** provided by GitLab Inc.

These are special virtual machines that run on GitLab's infrastructure and can build any project.

To enable the Shared Runners you have to go to your project's

Settings → CI/CD and click Enable shared runners.



# The Requirements File

The requirements file contains the number (and eventually the version) of the libraries that are needed to execute the code.

For example, to run the code in the examples of this lecture, the only required library that has to be installed is numpy.

Therefore, we have to have a file called “requirements.txt” which will contain just a single row:

*numpy*

# Check the Test Results

Project

Repository

Files

Commits

Branches

Tags

Contributors

Graph

Compare

Charts

Issues 0

Merge Requests 0

CI / CD

Collapse sidebar

Commit **f2f91854** authored 32 seconds ago by Ambra

Browse files

Options

add files

parents **master**

Pipeline #40466637 running with stage

Changes 3 Pipelines 1

Status	Pipeline	Commit	Stages
	#40466637 by  latest	<b>f2f91854</b> add files	

# Check the Test Results

Project

Repository

Issues0

Merge Requests0

CI / CD

Pipelines

**Jobs**

Schedules

Charts

Operations

Registry

Wiki

<< Collapse sidebar

running Job #135773294 triggered 26 seconds ago by Ambra

Running with gitlab-runner 11.5.0 (3afda6)

on docker-auto-scale ed2dce3a

Using Docker executor with image python:2 ...

Pulling docker image python:2 ...

...

Duration: 25 seconds

Timeout: 1h (from project)

Runner: shared-runners-manager-6.gitlab.com (#380987)

Cancel

Commit [dea1f789](#)

update .gitlabci

Pipeline #40466837 from master

test

→ test

# Check the Test Results

Project

Repository

Issues 0

Merge Requests 0

CI / CD

Pipelines

Jobs

Schedules

Charts

Operations

Registry

Wiki

<< Collapse sidebar

```
$ virtualenv venv
New python executable in /builds/Demontis/prova_isd/venv/bin/python
Installing setuptools, pip, wheel...
done.
$ source venv/bin/activate
$ pip install -rrequirements.txt
Collecting numpy (from -r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/de/37/
/fe7db552f4507f379d81dc78e58e05030a8941757b1f664517d581b5553/numpy-1.15.4-cp27-cp27mu-manylinux1_x86_64.whl
(13.8MB)
Installing collected packages: numpy
Successfully installed numpy-1.15.4
$ PYTHONPATH=. python tests/*
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

-----
Ran 3 tests in 0.002s

OK
Creating cache default...
WARNING: .cache/pip: no matching files
venv/: found 1781 matching files
Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners-cache/project/9968824/default
Created cache
Job succeeded
```

Duration: 57 seconds

Timeout: 1h (from project)

Runner: shared-runners-manager-4.gitlab.com (#44949)

Commit [ccaf821c](#)

update .gitlabci

✓ Pipeline #40467546 from master

test

→ ✓ test

# GitHub CI



# In Two Clicks...

Search or jump to... / Pull requests Issues Marketplace Explore

unica-isde / isde-testing Unwatch 1 Star 0 Fork 0

<> Code Issues Pull requests **Actions** Projects Wiki Security Insights Settings

Workflows **New workflow** All workflows

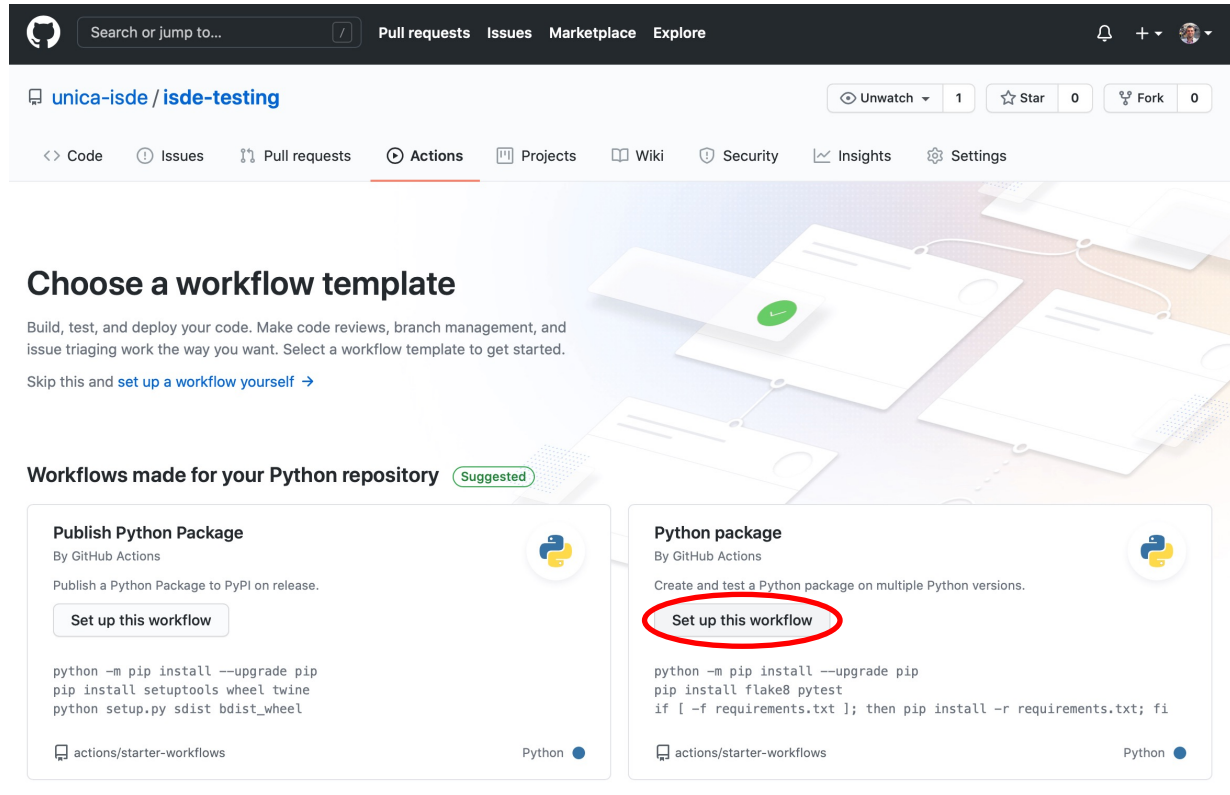
Python package

Filter workflows

3 results

	Event	Status	Branch	Actor
✓ Merge remote-tracking branch 'origin/main' into ... Python package #3: Commit 35b3fb9 pushed by bbiggio	main	3 minutes ago 30s	...	
✓ Update python-package.yml Python package #2: Commit 3c36050 pushed by bbiggio	main	5 minutes ago 25s	...	
✓ Create python-package.yml Python package #1: Commit 51485b7 pushed by bbiggio	main	6 minutes ago 32s	...	

# In Two Clicks...



The screenshot shows the GitHub repository page for `unica-isde/isde-testing`. The repository has 1 pull request, 0 stars, and 0 forks. The navigation bar includes links for Code, Issues, Pull requests, Actions (highlighted), Projects, Wiki, Security, Insights, and Settings. The main section is titled "Choose a workflow template" and provides instructions on how to build, test, and deploy code. Below this, there are two workflow templates for Python repositories. The first template, "Publish Python Package", is by GitHub Actions and includes a button "Set up this workflow". The second template, "Python package", is also by GitHub Actions and includes a button "Set up this workflow" which is circled in red. Both templates show the same code snippet: `python -m pip install --upgrade pip`, `pip install setuptools wheel twine`, and `python setup.py sdist bdist_wheel`. The source for both templates is `actions/starter-workflows` and they are categorized under Python.

Search or jump to... Pull requests Issues Marketplace Explore

unica-isde / isde-testing Unwatch 1 Star 0 Fork 0

Code Issues Pull requests **Actions** Projects Wiki Security Insights Settings

## Choose a workflow template

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow template to get started.

Skip this and [set up a workflow yourself](#) →

### Workflows made for your Python repository Suggested

#### Publish Python Package

By GitHub Actions

Publish a Python Package to PyPI on release.

[Set up this workflow](#)

```
python -m pip install --upgrade pip
pip install setuptools wheel twine
python setup.py sdist bdist_wheel
```

actions/starter-workflows Python

#### Python package

By GitHub Actions

Create and test a Python package on multiple Python versions.

[Set up this workflow](#)

```
python -m pip install --upgrade pip
pip install flake8 pytest
if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
```

actions/starter-workflows Python

# References

Unittest:

<https://docs.python.org/3/library/unittest.html>

GitLab CI:

[https://gitlab.com/help/ci/quick\\_start/README](https://gitlab.com/help/ci/quick_start/README)

GitHub *Building and Testing Python*:

<https://docs.github.com/en/free-pro-team@latest/actions/guides/building-and-testing-python#starting-with-the-python-workflow-template>