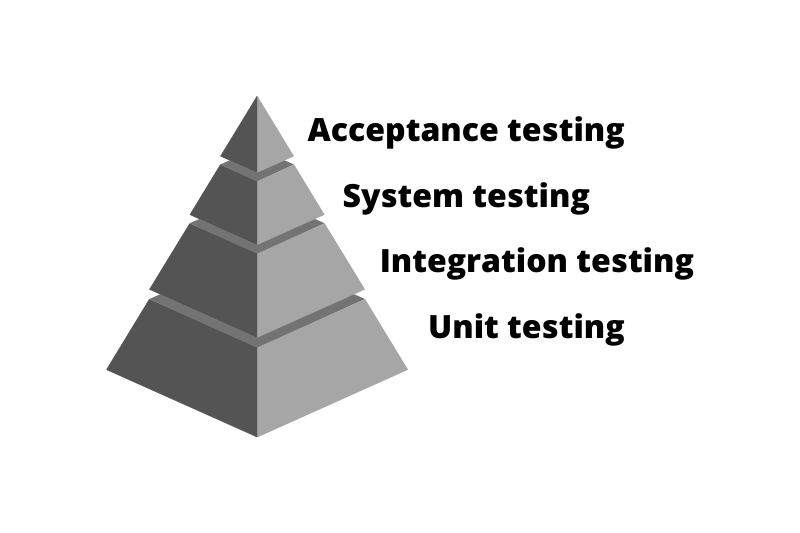
**How to Use Pytest for Unit Testing**  
Testing is a massive topic in software development. Before a software product gets into the hands of an end-user, it’s likely it would have gone through several tests such as integration tests, systems tests, and acceptance tests. The idea behind such vigorous testing is to ensure the behavior of the application is working as expected from the end-users standpoint. This approach to testing is known as behavior-driven development (BDD).

More recently, an interest in test-driven development (TDD) has grown significantly among developers. Diving into the depths of it may be a complex task for this article, but the general idea is that the traditional process of development and testing is reversed - you write your unit tests first, and then implement code changes until the tests pass.

In this article, we will focus on unit tests and, specifically, how to do them using a popular Python testing framework called Pytest.

**What are unit tests?**

Unit tests are a form of automated tests - this simply means that the test plan is executed by a script rather than manually by a human. They serve as the first level of software testing and are typically written in the form of functions that validate the behavior of various functionalities within a software program.



The levels of software testing

The idea behind these tests is to permit developers to isolate the smallest unit of code that makes logical sense and test that it behaves as expected. In other words, unit tests validate that a software program's single component works as the developers intended.

Ideally, these tests should be pretty small - the smaller they are, the better. One reason for building smaller tests is that the test will be more efficient since testing smaller units will enable the testing code to execute much faster. Another reason for testing smaller components is that it gives you greater insight into how the granular code behaves when merged.

**Why do we need unit tests?**

The global justification for why it’s essential to conduct unit tests is that developers must ensure the code they write meets the quality standards before permitting it to enter a production environment. However, several other factors contribute to the necessity of unit tests. Let’s dig deeper into some of those reasons.

**Preserves resources**

Carrying out unit tests help developers catch code bugs during the software construction stage, preventing them from transitioning deeper into the development lifecycle. This preserves resources since developers wouldn’t have to pay the cost of fixing bugs later in development - it also means end-users are less likely to have to deal with buggy code.

**Extra documentation**

Another great justification for conducting unit tests is that they serve as an extra layer of living documentation for your software product. Developers can simply refer to the unit tests for a holistic understanding of the overall system since they detail how the more minor components should behave.

**Confidence boost**

It’s extremely simple to make subtle mistakes in your code while writing up some functionality. However, most developers would agree it’s much better to identify the breaking points within a codebase before it’s put in a production environment: unit tests provide developers with this opportunity.

It’s fair to say that “code covered with unit tests can be considered more reliable than code that is not.” Future breaks in the code may be discovered much faster than code with no test coverage, saving time and money. Developers also benefit from extra documentation so they can understand the codebase quicker, and there is the added confidence of knowing that if they make a mistake in their code, it will be caught by the unit tests rather than by an end-user.

**Python testing frameworks**

Python has grown tremendously in popularity over the years. As part of Python’s growth, the number of testing frameworks has also increased, resulting in a wealth of tools available to help you test your Python code. Getting into the nitty-gritty of each tool is beyond the scope of this article, but we will touch on some of the most common Python testing frameworks available.

**unittest**

Unittest is a [**built-in Python framework**](https://docs.python.org/3/library/unittest.html) for unit testing. It was inspired by a unit testing framework called [**JUnit**](https://en.wikipedia.org/wiki/JUnit) from the Java programming language. Since it comes out of the box with the Python language, there are no extra modules to install, and most developers use it to [**begin learning about testing**](https://www.datacamp.com/tutorial/unit-testing-python).

**Pytest**

Pytest is possibly the most widely used Python testing framework around - this means it has a large community to support you whenever you get stuck. It’s an open-source framework that enables developers to write simple, compact test suites while supporting unit testing, functional testing, and API testing.

**doctest**

The **[doctest](https://docs.python.org/3/library/doctest.html" \t "_blank)** framework merges two core components of software engineering: documentation and testing. This functionality ensures that all software programs are thoroughly documented and tested to ensure they run as they should. **[doctest](https://docs.python.org/3/library/doctest.html" \t "_blank)** comes with Python’s standard library and is pretty straightforward to learn.

**nose2**

[**Nose2**](https://docs.nose2.io/), the successor to the [**nose**](https://nose.readthedocs.io/) regiment,  is essentially unittest with plugins. People often refer to [**nose2**](https://docs.nose2.io/) as “extended unit tests” or “unit tests with a plugin” due to its close ties to the Python built-in unit testing framework. Since it’s practically an extension of the unittest framework, [**nose2**](https://docs.nose2.io/) is incredibly easy to adopt for those familiar with unittest.

**Testify**

[**Testify**](https://github.com/Yelp/Testify), a Python framework for unit, integration, and system testing, is popularly known as the framework that was designed to replace unittest and nose. The framework is packed with extensive plugins and has quite a smooth learning curve if you’re already familiar with unittest.

**Hypothesis**

[**Hypothesis**](https://hypothesis.readthedocs.io/en/latest) enables developers to create unit tests that are simpler to write and are powerful when run. Since the framework is built to support data science projects, it helps to find edge cases that aren’t so apparent while you’re creating your tests by generating examples of inputs that align with specific properties you define.

For our tutorial, we will be using pytest. Check out the next section to see why you may wish to opt for Pytest over the others we’ve listed.

**Why use Pytest?**

Beyond its vast supportive community, pytest has several factors that make it one of the greatest tools to conduct your automated test suite in Python. Pytest’s philosophy and features are set up to make software testing a much better developer experience. One way the creators of Pytest achieved this goal is by significantly reducing the amount of code required to perform common tasks and making it possible to perform advanced tasks with extensive commands and plug-ins.

Some other reasons to use Pytest include the following:

**Easy to learn**

Pytest is extremely easy to learn: if you understand how Python’s assert keyword works, then you’re already well on your way to mastering the framework. Tests using pytest are Python functions with “test\_” prepended or “\_test” appended to the function's name - although you can use a class to group multiple tests. Overall, the learning curve for pytest is much shallower than the likes of unittest since you’re not required to learn any new constructs.

**Test filtering**

You may not want to run all of your tests with each execution - this may be the case as your test suite grows. Sometimes, you may wish to isolate a few tests on a new feature to get rapid feedback while you’re developing, then run the full suite once you’re confident everything is functioning as planned. Pytest has three ways you could isolate tests: 1) name-based filtering, which tells pytest only to run the tests whose names match the pattern provided 2) directory scoping, which is a default setting that tells pytest to only run tests that are in or under the current directory and 3) test categorization which allows you to define categories for tests that pytest should include or exclude.

**Parameterization**

Pytest has a built-in decorator called parametrize that enables the parametrization of arguments for a test function. Thus, if the functions you’re testing process data or performs a generic transformation, you are not required to write several similar tests. We will cover more on [**parametrization**](https://docs.google.com/document/d/1I6qAzZyzpHKqebj5pJTaReliGXFbYjRmYVhDBMQNbKM/edit?pli=1#heading=h.d8rewh1zp4e0) later in the article.

We will stop here, but the list of why pytest is a great tooling option for your automated test suite goes on.

**Pytest vs unittest**

Despite all of the reasons we’ve covered above, one may still dispute the idea of using pytest for the simple fact that it’s a third-party framework - “what’s the point of installing a framework if there’s already one built-in?” It’s a good argument, but to cover our back in that dispute, we’ll provide you with some things to consider.

**Note**: If you’re already sold on pytest, skip to the next section where we get to grips with how to use the framework.

**Less boilerplate**

Unittest requires developers to create classes derived from the **[TestCase](https://docs.python.org/3/library/unittest.html" \t "_blank)** module and then define the test cases as methods in the class.

"""

An example test case with unittest.

See: https://docs.python.org/3/library/unittest.html

"""

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)

This code defines a test case using the unittest module in Python. The unittest module provides a framework for writing and running tests in Python.

The TestStringMethods class inherits from unittest.TestCase, which provides a set of assertion methods to check for expected outcomes.

The test\_upper, test\_isupper, and test\_split methods are test cases that check various string methods.

In test\_upper, the assertEqual method checks if the result of 'foo'.upper() is equal to 'FOO'.

In test\_isupper, the assertTrue and assertFalse methods check if 'FOO' and 'Foo' are uppercase or not.

In test\_split, the assertEqual method checks if the result of s.split() is equal to ['hello', 'world']. The with statement and assertRaises method check if s.split(2) raises a TypeError when the separator is not a string.

Overall, this code demonstrates how to write and run test cases using the unittest module in Python.

Pytest, on the other hand, only requires you to define a function with “test\_” prepended and use the assert conditions inside them.

"""

An example test case with pytest.

See: https://docs.pytest.org/en/6.2.x/index.html

"""

# content of test\_sample.py

def inc(x):

return x + 1

def test\_answer():

assert inc(3) == 5

This code defines a simple test case using the pytest framework. The inc function takes a number x and returns x + 1. The test\_answer function asserts that calling inc(3) should return 5.

When this test is run using pytest, it will pass if the assertion is true and fail if the assertion is false. This is a simple example of how to use pytest to write and run tests for your code.

Notice the difference in the amount of code required; unittest has a significant amount of boilerplate code required, which serves as a minimum requirement for any test you’d like to perform. This means it’s highly likely you’ll end up writing the same code several times over. Pytest, on the other hand, has rich inbuilt features that simplify this workflow by reducing the amount of code required to write test cases.

**Output**

The outputs provided by each framework is extremely different. Here’s an example pytest execution:

"""

See: https://docs.pytest.org/en/6.2.x/index.html

"""

$ pytest

=========================== test session starts ============================

platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y

cachedir: $PYTHON\_PREFIX/.pytest\_cache

rootdir: $REGENDOC\_TMPDIR

collected 1 item

================================= FAILURES =================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_answer \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_answer():

> assert inc(3) == 5

E assert 4 == 5

E + where 4 = inc(3)

test\_sample.py:6: AssertionError

========================= short test summary info ==========================

FAILED test\_sample.py::test\_answer - assert 4 == 5

============================ 1 failed in 0.12s =============================

The test case above failed, but notice how detailed the breakdown of the failure is. This makes it easier for developers to identify where the bugs in their code exist, which is super helpful when debugging. As an added bonus, there’s also an overall status report for the test suite, which tells us the number of tests that failed and how long it took.

Let’s take a look at a failed test case example with unittest.

import unittest

def square(n):

return n\*n

def cube(n):

return n\*n\*n

class TestCase(unittest.TestCase):

def test\_square(self):

self.asserEquals(square(4), 16)

def test\_cube(self):

self.assertEquals(cube(4), 16)

This code defines two functions, square and cube, which calculate the square and cube of a given number, respectively. It also imports the unittest module, which provides a framework for writing and running tests.

The TestCase class is defined, which inherits from unittest.TestCase. This class contains two methods, test\_square and test\_cube, which are test cases for the square and cube functions, respectively.

Each test case uses the assertEqual method to check that the output of the function matches the expected value. If the test fails, an error message will be displayed.

To run the tests, the unittest module provides a test runner that can be invoked from the command line or from within an IDE. When the tests are run, the output will indicate whether each test passed or failed.

When we run the script, we get the following output:

---------------------------------------------------------------

Ran 2 tests in 0.001s

FAILED (errors=1)

This code snippet is not a complete code block, but rather a result of running a test suite. The output indicates that two tests were run and one of them failed, resulting in an error. The time taken to run the tests was 0.001 seconds. It is not possible to determine the programming language used based on this output alone.

The output above tells us that two tests were run in 0.001s and one failed, but not much else. Ultimately, pytest provides much more informative feedback, which comes in handy when it’s time to debug.

All in all, both pytest and unittest are great tools to use for automated testing in Python. Several Python developers may lean more towards pytest over its counterparts due to its compactness and efficiency. It’s also extremely easy to adopt and there are several features you could use to build an effective test suite.

Now for the main part of this article. We’ve discussed what unit testing is and why pytest is a great tool for automated testing in Python. Now let’s have a look at how to use the tool.

**Pytest tutorial**

Let’s see how this Python testing framework we’ve been going on about works.

The first step is installing the package, which can be done with a simple pip command.

Note: The creators of pytest recommend you use **[venv](https://docs.python.org/3/library/venv.html" \t "_blank)** for development and [**pip**](https://pypi.org/project/pip) for installing your application, dependencies, and pytest itself.

pip install -U pytest

Next, check to see the framework has been installed using the following command:

>>>> pytest --version

pytest 7.1.2

Everything is installed. You’re now ready to start running some tests.

**Creating a simple test**

Creating a test is simple with Pytest. To demonstrate this functionality, we’ve created a script called [**calcualte\_age.py**](https://github.com/kurtispykes/Machine-Learning/blob/main/testing/calculate_age.py). This script has only one function, get\_age, which is responsible for calculating a user's age, given their date of birth.

import datetime

def get\_age(yyyy:int, mm:int, dd:int) -> int:

dob = datetime.date(yyyy, mm, dd)

today = datetime.date.today()

age = round((today - dob).days / 365.25)

return age

This code imports the datetime module, which provides classes for working with dates and times.

The get\_age function takes three integer arguments: yyyy for the year of birth, mm for the month of birth, and dd for the day of birth. The function returns an integer representing the age of the person based on their date of birth.

Inside the function, the datetime.date function is used to create a dob object representing the date of birth. The datetime.date.today() function is used to create a today object representing the current date.

The age variable is calculated by subtracting the dob object from the today object, which results in a datetime.timedelta object representing the difference between the two dates. The .days attribute of the timedelta object is divided by 365.25 (to account for leap years) and rounded to the nearest integer using the round function.

Finally, the age variable is returned as an integer.

Pytest will execute all the python files that have the name test\_ prepended or \_test appended to the name of the script. To be more specific, pytest follows the following conventions for test discovery [source: [**documentation**](https://docs.pytest.org/en/6.2.x/goodpractices.html#test-discovery)]:

* Given no arguments are specified, pytest collection would begin in **[testpaths](https://docs.pytest.org/en/6.2.x/reference.html" \l "confval-testpaths" \t "_blank)** if they are configured: testpaths are a list of directories pytest will search when no specific directories, files, or test ids are provided.
* Pytest would then recurse into directories unless you’ve told it not to by setting **[norecursedirs](https://docs.pytest.org/en/6.2.x/reference.html" \l "confval-norecursedirs" \t "_blank)**; It’s searching for files that begin with test\_\*.py or end in \*\_test.py
* In those files, pytest would collect test items in the following order:
  + Prefixed test functions or methods outside of class
  + Prefixed test functions or methods inside Test prefixed test classes that do not have an \_\_init\_\_ method.

We have not specified any arguments, but we have created another script in the same directory called [**test\_calculate\_age.py**](https://github.com/kurtispykes/Machine-Learning/blob/main/testing/test_calculate_age.py): thus, when the directories are recursed the test will be discovered.  In this script, we have a single test, test\_get\_age, to validate our function is working accordingly.

**Note**: You may decide to put your tests into an extra directory outside of your application which is a good idea if you have several functional tests or you want to keep testing code and application code separate for some other reason.

from calculate\_age import get\_age

def test\_get\_age():

# Given.

yyyy, mm, dd = map(int, "1996/07/11".split(""))

# When.

age = get\_age(yyyy, mm, dd)

# Then.

assert age == 26

This code is written in Python.

The code imports a function called get\_age from a module called calculate\_age.

Then, a function called test\_get\_age is defined. This function tests the get\_age function by checking if the age calculated from the given birthdate (1996/07/11) is equal to 26.

In the Given section, the birthdate is split into year, month, and day using the split method with an empty string as the separator. The resulting strings are then converted to integers using the map function and int constructor.

In the When section, the get\_age function is called with the year, month, and day as arguments, and the resulting age is stored in the age variable.

In the Then section, an assertion is made that the age variable is equal to 26. If the assertion fails, an error will be raised.

Overall, this code tests the get\_age function to ensure that it correctly calculates the age from a given birthdate.

To execute the test, run the following command from the command prompt:

py -m pytest

This code runs the pytest module using the Python interpreter. The -m flag specifies that the module should be run as a script. The pytest module is a testing framework for Python that allows developers to write and run tests for their code. Running this command will execute all the tests in the current directory and its subdirectories.