**Testing your code**

Remember when you ran your application and used it for the first time? Did you check the features and experiment using them? That’s known as **exploratory testing** and is a form of manual testing.

Exploratory testing is a form of testing that is done without a plan. In an exploratory test, you’re just exploring the application.

To have a complete set of manual tests, all you need to do is make a list of all the features your application has, the different types of input it can accept, and the expected results. Now, every time you make a change to your code, you need to go through every single item on that list and check it.

This is where automated testing comes in. Automated testing is the execution of your test plan (the parts of your application you want to test, the order in which you want to test them, and the expected responses) by a script instead of a human. Python already comes with a set of tools and libraries to help you create automated tests for your application.

Unit Tests vs. Integration Tests

The world of testing has no shortage of terminology, and now that you know the difference between automated and manual testing, it’s time to go a level deeper.

Think of how you might test the lights on a car. You would turn on the lights (known as the **test step**) and go outside the car or ask a friend to check that the lights are on (known as the **test assertion**). Testing multiple components is known as **integration testing**.

Think of all the things that need to work correctly in order for a simple task to give the right result. These components are like the parts to your application, all of those classes, functions, and modules you’ve written.

A major challenge with integration testing is when an integration test doesn’t give the right result. It’s very hard to diagnose the issue without being able to isolate which part of the system is failing. If the lights didn’t turn on, then maybe the bulbs are broken. Is the battery dead? What about the alternator? Is the car’s computer failing?

If you have a fancy modern car, it will tell you when your light bulbs have gone. It does this using a form of **unit test**.

A unit test is a smaller test, one that checks that a single component operates in the right way. A unit test helps you to isolate what is broken in your application and fix it faster.

You have just seen two types of tests:

1. An integration test checks that components in your application operate with each other.
2. A unit test checks a small component in your application.

You can write both integration tests and unit tests in Python. To write a unit test for the built-in function sum(), you would check the output of sum() against a known output.

For example, here’s how you check that the sum() of the numbers (1, 2, 3) equals 6:

assert sum([1, 2, 3]) == 6, "Should be 6"

python test\_sum.py

Everything passed

AssertionError:

>>>

>>> assert sum([1, 1, 1]) == 6, "Should be 6"

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AssertionError: Should be 6

In the REPL, you are seeing the raised AssertionError because the result of sum() does not match 6.

Instead of testing on the REPL, you’ll want to put this into a new Python file called test\_sum.py and execute it again:

Now you have written a **test case**, an assertion, and an entry point (the command line). You can now execute this at the command line:

$ python test\_sum.py

Everything passed

You can see the successful result, Everything passed.

In Python, sum() accepts any iterable as its first argument. You tested with a list. Now test with a tuple as well. Create a new file called test\_sum\_2.py with the following code:

def test\_sum():

assert sum([1, 2, 3]) == 6, "Should be 6"

def test\_sum\_tuple():

assert sum((1, 2, 2)) == 6, "Should be 6"

if \_\_name\_\_ == "\_\_main\_\_":

test\_sum()

test\_sum\_tuple()

print("Everything passed")

When you execute test\_sum\_2.py, the script will give an error because the sum() of (1, 2, 2) is 5, not 6. The result of the script gives you the error message, the line of code, and the traceback:

$ python test\_sum\_2.py

Traceback (most recent call last):

File "test\_sum\_2.py", line 9, in <module>

test\_sum\_tuple()

File "test\_sum\_2.py", line 5, in test\_sum\_tuple

assert sum((1, 2, 2)) == 6, "Should be 6"

AssertionError: Should be 6

Here you can see how a mistake in your code gives an error on the console with some information on where the error was and what the expected result was.

Writing tests in this way is okay for a simple check, but what if more than one fails? This is where test runners come in. The test runner is a special application designed for running tests, checking the output, and giving you tools for debugging and diagnosing tests and applications.

### Choosing a Test Runner

There are many test runners available for Python. The one built into the Python standard library is called unittest. In this tutorial, you will be using unittest test cases and the unittest test runner. The principles of unittest are easily portable to other frameworks. The three most popular test runners are:

* unittest
* nose or nose2
* pytest

Choosing the best test runner for your requirements and level of experience is important.

#### unittest

unittest has been built into the Python standard library since version 2.1. You’ll probably see it in commercial Python applications and open-source projects.

unittest contains both a testing framework and a test runner. unittest has some important requirements for writing and executing tests.

unittest requires that:

* You put your tests into classes as methods
* You use a series of special assertion methods in the unittest.TestCase class instead of the built-in assert statement

To convert the earlier example to a unittest test case, you would have to:

1. [Import](https://realpython.com/absolute-vs-relative-python-imports/) unittest from the standard library
2. Create a class called TestSum that inherits from the TestCase class
3. Convert the test functions into methods by adding self as the first argument
4. Change the assertions to use the self.assertEqual() method on the TestCase class
5. Change the command-line entry point to call unittest.main()

Follow those steps by creating a new file test\_sum\_unittest.py with the following code:

import unittest

class TestSum(unittest.TestCase):

def test\_sum(self):

self.assertEqual(sum([1, 2, 3]), 6, "Should be 6")

def test\_sum\_tuple(self):

self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

If you execute this at the command line, you’ll see one success (indicated with .) and one failure (indicated with F):

$ python test\_sum\_unittest.py

.F

======================================================================

FAIL: test\_sum\_tuple (\_\_main\_\_.TestSum)

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

Traceback (most recent call last):

File "test\_sum\_unittest.py", line 9, in test\_sum\_tuple

self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")

AssertionError: Should be 6

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

Ran 2 tests in 0.001s

FAILED (failures=1)