11

TESTING YOUR CODE

When you write a function or a class, you can also write tests for that code. Testing proves that your code works as it's supposed to in response to all the input types it's designed to receive. When you write tests, you can be confident that your code will work correctly as more people begin to use your programs. You'll also be able to test new code as you add it to make sure your changes don't break your program's existing behavior. Every programmer makes mistakes, so every programmer must test their code often, catching problems before users encounter them.

In this chapter you'll learn to test your code using tools in Python's `unittest` module. You'll learn to build a test case and check that a set of inputs results in the output you want. You'll see what a passing test looks like and what a failing test looks like, and you'll learn how a failing test can help you improve your code. You'll learn to test functions and classes, and you'll start to understand how many tests to write for a project.

Testing a Function

To learn about testing, we need code to test. Here's a simple function that takes in a first and last name, and returns a neatly formatted full name:

name_function.py

```
def get_formatted_name(first, last):
    """Generate a neatly formatted full name."""
```

```
    full_name = f"{first} {last}"
    return full_name.title()
```

The function `get_formatted_name()` combines the first and last name with a space in between to complete a full name, and then capitalizes and returns the full name. To check that `get_formatted_name()` works, let's make a program that uses this function. The program *names.py* lets users enter a first and last name, and see a neatly formatted full name:

names.py

```
from name_function import get_formatted_name

print("Enter 'q' at any time to quit.")
while True:
    first = input("\nPlease give me a first name: ")
    if first == 'q':
        break
    last = input("Please give me a last name: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print(f"\tNeatly formatted name: {formatted_name}.")
```

This program imports `get_formatted_name()` from *name_function.py*. The user can enter a series of first and last names, and see the formatted full names that are generated:

```
Enter 'q' at any time to quit.

Please give me a first name: janis
Please give me a last name: joplin
        Neatly formatted name: Janis Joplin.

Please give me a first name: bob
```

Please give me a last name: **dylan**

    Neatly formatted name: Bob Dylan.

Please give me a first name: **q**

We can see that the names generated here are correct. But let's say we want to modify `get_formatted_name()` so it can also handle middle names. As we do so, we want to make sure we don't break the way the function handles names that have only a first and last name. We could test our code by running *names.py* and entering a name like `Janis Joplin` every time we modify `get_formatted_name()`, but that would become tedious. Fortunately, Python provides an efficient way to automate the testing of a function's output. If we automate the testing of `get_formatted_name()`, we can always be confident that the function will work when given the kinds of names we've written tests for.

Unit Tests and Test Cases

The module `unittest` from the Python standard library provides tools for testing your code. A *unit test* verifies that one specific aspect of a function's behavior is correct. A *test case* is a collection of unit tests that together prove that a function behaves as it's supposed to, within the full range of situations you expect it to handle. A good test case considers all the possible kinds of input a function could receive and includes tests to represent each of these situations. A test case with *full coverage* includes a full range of unit tests covering all the possible ways you can use a function. Achieving full coverage on a large project can be daunting. It's often good enough to write tests for your code's critical behaviors and then aim for full coverage only if the project starts to see widespread use.

A Passing Test

The syntax for setting up a test case takes some getting used to, but once you've set up the test case it's straightforward to add more unit tests for your functions. To write a test case for a function, import the `unittest` module and the function you want to test. Then create a class that inherits

from `unittest.TestCase`, and write a series of methods to test different aspects of your function's behavior.

Here's a test case with one method that verifies that the function `get_formatted_name()` works correctly when given a first and last name:

test_name_function.py

```
import unittest
from name_function import get_formatted_name

❶ class NamesTestCase(unittest.TestCase):
    """Tests for 'name_function.py'."""

    def test_first_last_name(self):
        """Do names like 'Janis Joplin' work?"""
❷       formatted_name = get_formatted_name('janis', 'joplin')
❸       self.assertEqual(formatted_name, 'Janis Joplin')

❹ if __name__ == '__main__':
    unittest.main()
```

First, we import `unittest` and the function we want to test, `get_formatted_name()`. At ❶ we create a class called `NamesTestCase`, which will contain a series of unit tests for `get_formatted_name()`. You can name the class anything you want, but it's best to call it something related to the function you're about to test and to use the word *Test* in the class name. This class must inherit from the class `unittest.TestCase` so Python knows how to run the tests you write.

`NamesTestCase` contains a single method that tests one aspect of `get_formatted_name()`. We call this method `test_first_last_name()` because we're verifying that names with only a first and last name are formatted correctly. Any method that starts with `test_` will be run automatically when we run *test_name_function.py*. Within this test method, we call the func-

tion we want to test. In this example we call `get_formatted_name()` with the arguments `'janis'` and `'joplin'`, and assign the result to `formatted_name` ❷.

At ❸ we use one of `unittest`'s most useful features: an *assert* method. Assert methods verify that a result you received matches the result you expected to receive. In this case, because we know `get_formatted_name()` is supposed to return a capitalized, properly spaced full name, we expect the value of `formatted_name` to be `Janis Joplin`. To check if this is true, we use `unittest`'s `assertEqual()` method and pass it `formatted_name` and `'Janis Joplin'`. The line

```
self.assertEqual(formatted_name, 'Janis Joplin')
```

says, "Compare the value in `formatted_name` to the string `'Janis Joplin'`. If they are equal as expected, fine. But if they don't match, let me know!"

We're going to run this file directly, but it's important to note that many testing frameworks import your test files before running them. When a file is imported, the interpreter executes the file as it's being imported. The `if` block at ❹ looks at a special variable, `__name__`, which is set when the program is executed. If this file is being run as the main program, the value of `__name__` is set to `'__main__'`. In this case, we call `unittest.main()`, which runs the test case. When a testing framework imports this file, the value of `__name__` won't be `'__main__'` and this block will not be executed.

When we run *test_name_function.py*, we get the following output:

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

OK
```

The dot on the first line of output tells us that a single test passed. The next line tells us that Python ran one test, and it took less than 0.001 sec-

onds to run. The final OK tells us that all unit tests in the test case passed.

This output indicates that the function `get_formatted_name()` will always work for names that have a first and last name unless we modify the function. When we modify `get_formatted_name()`, we can run this test again. If the test case passes, we know the function will still work for names like Janis Joplin.

A Failing Test

What does a failing test look like? Let's modify `get_formatted_name()` so it can handle middle names, but we'll do so in a way that breaks the function for names with just a first and last name, like Janis Joplin.

Here's a new version of `get_formatted_name()` that requires a middle name argument:

name_function.py

```
def get_formatted_name(first, middle, last):
    """Generate a neatly formatted full name."""
    full_name = f"{first} {middle} {last}"
    return full_name.title()
```

This version should work for people with middle names, but when we test it, we see that we've broken the function for people with just a first and last name. This time, running the file *test_name_function.py* gives this output:

```
❶ E
  ======================================================================
❷ ERROR: test_first_last_name (__main__.NamesTestCase)
  ----------------------------------------------------------------------
❸ Traceback (most recent call last):
    File "test_name_function.py", line 8, in test_first_last_name
      formatted_name = get_formatted_name('janis', 'joplin')
  TypeError: get_formatted_name() missing 1 required positional argu-
```

ment: 'last'

----

❹ Ran 1 test in 0.000s

❺ FAILED (errors=1)

There's a lot of information here because there's a lot you might need to know when a test fails. The first item in the output is a single `E` ❶, which tells us one unit test in the test case resulted in an error. Next, we see that `test_first_last_name()` in `NamesTestCase` caused an error ❷. Knowing which test failed is critical when your test case contains many unit tests. At ❸ we see a standard traceback, which reports that the function call `get_formatted_name('janis', 'joplin')` no longer works because it's missing a required positional argument.

We also see that one unit test was run ❹. Finally, we see an additional message that the overall test case failed and that one error occurred when running the test case ❺. This information appears at the end of the output so you see it right away; you don't need to scroll up through a long output listing to find out how many tests failed.

Responding to a Failed Test

What do you do when a test fails? Assuming you're checking the right conditions, a passing test means the function is behaving correctly and a failing test means there's an error in the new code you wrote. So when a test fails, don't change the test. Instead, fix the code that caused the test to fail. Examine the changes you just made to the function, and figure out how those changes broke the desired behavior.

In this case `get_formatted_name()` used to require only two parameters: a first name and a last name. Now it requires a first name, middle name, and last name. The addition of that mandatory middle name parameter broke the desired behavior of `get_formatted_name()`. The best option here is to make the middle name optional. Once we do, our test for names like `Janis`

`Joplin` should pass again, and we should be able to accept middle names as well. Let's modify `get_formatted_name()` so middle names are optional and then run the test case again. If it passes, we'll move on to making sure the function handles middle names properly.

To make middle names optional, we move the parameter `middle` to the end of the parameter list in the function definition and give it an empty default value. We also add an `if` test that builds the full name properly, depending on whether or not a middle name is provided:

name_function.py

```
def get_formatted_name(first, last, middle=''):
    """Generate a neatly formatted full name."""
    if middle:
        full_name = f"{first} {middle} {last}"
    else:
        full_name = f"{first} {last}"
    return full_name.title()
```

In this new version of `get_formatted_name()`, the middle name is optional. If a middle name is passed to the function, the full name will contain a first, middle, and last name. Otherwise, the full name will consist of just a first and last name. Now the function should work for both kinds of names. To find out if the function still works for names like `Janis Joplin`, let's run *test_name_function.py* again:

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

OK
```

The test case passes now. This is ideal; it means the function works for names like `Janis Joplin` again without us having to test the function manu-

ally. Fixing our function was easy because the failed test helped us identify the new code that broke existing behavior.

Adding New Tests

Now that we know `get_formatted_name()` works for simple names again, let's write a second test for people who include a middle name. We do this by adding another method to the class `NamesTestCase`:

test_name_function.py

```
--snip--

class NamesTestCase(unittest.TestCase):
    """Tests for 'name_function.py'."""

    def test_first_last_name(self):
        --snip--

    def test_first_last_middle_name(self):
        """Do names like 'Wolfgang Amadeus Mozart' work?"""
❶       formatted_name = get_formatted_name(
            'wolfgang', 'mozart', 'amadeus')
        self.assertEqual(formatted_name, 'Wolfgang Amadeus Mozart')

if __name__ == '__main__':
    unittest.main()
```

We name this new method `test_first_last_middle_name()`. The method name must start with *test_* so the method runs automatically when we run *test_name_function.py*. We name the method to make it clear which behavior of `get_formatted_name()` we're testing. As a result, if the test fails, we know right away what kinds of names are affected. It's fine to have long method names in your `TestCase` classes. They need to be descriptive so you can make sense of the output when your tests fail, and because Python

calls them automatically, you'll never have to write code that calls these methods.

To test the function, we call `get_formatted_name()` with a first, last, and middle name ❶, and then we use `assertEqual()` to check that the returned full name matches the full name (first, middle, and last) that we expect. When we run *test_name_function.py* again, both tests pass:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Great! We now know that the function still works for names like `Janis Joplin`, and we can be confident that it will work for names like `Wolfgang Amadeus Mozart` as well.

TRY IT YOURSELF

11-1. City, Country: Write a function that accepts two parameters: a city name and a country name. The function should return a single string of the form `City, Country`, such as `Santiago, Chile`. Store the function in a module called *city_functions.py*.

Create a file called *test_cities.py* that tests the function you just wrote (remember that you need to import `unittest` and the function you want to test). Write a method called `test_city_country()` to verify that calling your function with values such as `'santiago'` and `'chile'` results in the correct string. Run *test_cities.py*, and make sure `test_city_country()` passes.

11-2. Population: Modify your function so it requires a third parameter, `population`. It should now return a single string of the form `City, Country - population xxx`, such as `Santiago, Chile - population 5000000`. Run *test_cities.py* again. Make sure `test_city_country()` fails this time.

Modify the function so the `population` parameter is optional. Run *test_cities.py* again, and make sure `test_city_country()` passes again.

Write a second test called `test_city_country_population()` that verifies you can call your function with the values `'santiago'`, `'chile'`, and `'population=5000000'`. Run *test_cities.py* again, and make sure this new test passes.

Testing a Class

In the first part of this chapter, you wrote tests for a single function. Now you'll write tests for a class. You'll use classes in many of your own programs, so it's helpful to be able to prove that your classes work correctly. If you have passing tests for a class you're working on, you can be confident that improvements you make to the class won't accidentally break its current behavior.

A Variety of Assert Methods

Python provides a number of assert methods in the `unittest.TestCase` class. As mentioned earlier, assert methods test whether a condition you believe is true at a specific point in your code is indeed true. If the condition is true as expected, your assumption about how that part of your program behaves is confirmed; you can be confident that no errors exist. If the condition you assume is true is actually not true, Python raises an exception.

Table 11-1 describes six commonly used assert methods. With these methods you can verify that returned values equal or don't equal expected values, that values are `True` or `False`, and that values are `in` or `not in` a given list. You can use these methods only in a class that inherits from `unittest.TestCase`, so let's look at how we can use one of these methods in the context of testing an actual class.

Table 11-1: Assert Methods Available from the `unittest` Module

Method	Use
`assertEqual(a, b)`	Verify that `a == b`
`assertNotEqual(a, b)`	Verify that `a != b`
`assertTrue(x)`	Verify that `x` is `True`
`assertFalse(x)`	Verify that `x` is `False`
`assertIn(item, list)`	Verify that *`item`* is in *`list`*
`assertNotIn(item, list)`	Verify that *`item`* is not in *`list`*

A Class to Test

Testing a class is similar to testing a function—much of your work involves testing the behavior of the methods in the class. But there are a few differences, so let's write a class to test. Consider a class that helps administer anonymous surveys:

survey.py

```
class AnonymousSurvey:
    """Collect anonymous answers to a survey question."""

❶   def __init__(self, question):
        """Store a question, and prepare to store responses."""
        self.question = question
        self.responses = []

❷   def show_question(self):
        """Show the survey question."""
        print(self.question)
```

```
❸    def store_response(self, new_response):
        """Store a single response to the survey."""
        self.responses.append(new_response)

❹    def show_results(self):
        """Show all the responses that have been given."""
        print("Survey results:")
        for response in self.responses:
            print(f"- {response}")
```

This class starts with a survey question that you provide ❶ and includes an empty list to store responses. The class has methods to print the survey question ❷, add a new response to the response list ❸, and print all the responses stored in the list ❹. To create an instance from this class, all you have to provide is a question. Once you have an instance representing a particular survey, you display the survey question with `show_question()`, store a response using `store_response()`, and show results with `show_results()`.

To show that the `AnonymousSurvey` class works, let's write a program that uses the class:

language_survey.py

```
from survey import AnonymousSurvey

# Define a question, and make a survey.
question = "What language did you first learn to speak?"
my_survey = AnonymousSurvey(question)

# Show the question, and store responses to the question.
my_survey.show_question()
print("Enter 'q' at any time to quit.\n")
while True:
    response = input("Language: ")
    if response == 'q':
```

```
        break
    my_survey.store_response(response)

# Show the survey results.
print("\nThank you to everyone who participated in the survey!")
my_survey.show_results()
```

This program defines a question (`"What language did you first learn to speak?"`) and creates an `AnonymousSurvey` object with that question. The program calls `show_question()` to display the question and then prompts for responses. Each response is stored as it is received. When all responses have been entered (the user inputs `q` to quit), `show_results()` prints the survey results:

```
What language did you first learn to speak?
Enter 'q' at any time to quit.

Language: English
Language: Spanish
Language: English
Language: Mandarin
Language: q

Thank you to everyone who participated in the survey!
Survey results:
- English
- Spanish
- English
- Mandarin
```

This class works for a simple anonymous survey. But let's say we want to improve `AnonymousSurvey` and the module it's in, `survey`. We could allow each user to enter more than one response. We could write a method to list only unique responses and to report how many times each response was given. We could write another class to manage nonanonymous surveys.

Implementing such changes would risk affecting the current behavior of the class `AnonymousSurvey`. For example, it's possible that while trying to allow each user to enter multiple responses, we could accidentally change how single responses are handled. To ensure we don't break existing behavior as we develop this module, we can write tests for the class.

Testing the AnonymousSurvey Class

Let's write a test that verifies one aspect of the way `AnonymousSurvey` behaves. We'll write a test to verify that a single response to the survey question is stored properly. We'll use the `assertIn()` method to verify that the response is in the list of responses after it's been stored:

test_survey.py

```
import unittest
from survey import AnonymousSurvey

❶ class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey"""
❷   def test_store_single_response(self):
        """Test that a single response is stored properly."""
        question = "What language did you first learn to speak?"
❸       my_survey = AnonymousSurvey(question)
        my_survey.store_response('English')
❹       self.assertIn('English', my_survey.responses)

if __name__ == '__main__':
    unittest.main()
```

We start by importing the `unittest` module and the class we want to test, `AnonymousSurvey`. We call our test case `TestAnonymousSurvey`, which again inherits from `unittest.TestCase` ❶. The first test method will verify that when we store a response to the survey question, the response ends up in the survey's list of responses. A good descriptive name for this method is `test_store_single_response()` ❷. If this test fails, we'll know from the method

name shown in the output of the failing test that there was a problem storing a single response to the survey.

To test the behavior of a class, we need to make an instance of the class. At ❸ we create an instance called `my_survey` with the question `"What language did you first learn to speak?"` We store a single response, `English`, using the `store_response()` method. Then we verify that the response was stored correctly by asserting that `English` is in the list `my_survey.responses` ❹.

When we run *test_survey.py*, the test passes:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

This is good, but a survey is useful only if it generates more than one response. Let's verify that three responses can be stored correctly. To do this, we add another method to `TestAnonymousSurvey`:

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey"""

    def test_store_single_response(self):
        --snip--

    def test_store_three_responses(self):
        """Test that three individual responses are stored properly."""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
❶       responses = ['English', 'Spanish', 'Mandarin']
        for response in responses:
```

```
            my_survey.store_response(response)

❷       for response in responses:
            self.assertIn(response, my_survey.responses)

  if __name__ == '__main__':
      unittest.main()
```

We call the new method `test_store_three_responses()`. We create a survey object just like we did in `test_store_single_response()`. We define a list containing three different responses ❶, and then we call `store_response()` for each of these responses. Once the responses have been stored, we write another loop and assert that each response is now in `my_survey.responses` ❷.

When we run *test_survey.py* again, both tests (for a single response and for three responses) pass:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

This works perfectly. However, these tests are a bit repetitive, so we'll use another feature of `unittest` to make them more efficient.

The setUp() Method

In *test_survey.py* we created a new instance of `AnonymousSurvey` in each test method, and we created new responses in each method. The `unittest.TestCase` class has a `setUp()` method that allows you to create these objects once and then use them in each of your test methods. When you include a `setUp()` method in a `TestCase` class, Python runs the `setUp()` method before running each method starting with *test_*. Any objects created in the `setUp()` method are then available in each test method you write.

Let's use `setUp()` to create a survey instance and a set of responses that can be used in `test_store_single_response()` and `test_store_three_responses()`:

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey."""

    def setUp(self):
        """
        Create a survey and a set of responses for use in all test methods.
        """
        question = "What language did you first learn to speak?"
❶       self.my_survey = AnonymousSurvey(question)
❷       self.responses = ['English', 'Spanish', 'Mandarin']

    def test_store_single_response(self):
        """Test that a single response is stored properly."""
        self.my_survey.store_response(self.responses[0])
        self.assertIn(self.responses[0], self.my_survey.responses)

    def test_store_three_responses(self):
        """Test that three individual responses are stored properly."""
        for response in self.responses:
            self.my_survey.store_response(response)
        for response in self.responses:
            self.assertIn(response, self.my_survey.responses)

if __name__ == '__main__':
    unittest.main()
```

The method `setUp()` does two things: it creates a survey instance ❶, and it creates a list of responses ❷. Each of these is prefixed by `self`, so they can be used anywhere in the class. This makes the two test methods simpler, because neither one has to make a survey instance or a response.

The method `test_store_single_response()` verifies that the first response in `self.responses`—`self.responses[0]`—can be stored correctly, and `test_store_three_responses()` verifies that all three responses in `self.responses` can be stored correctly.

When we run *test_survey.py* again, both tests still pass. These tests would be particularly useful when trying to expand `AnonymousSurvey` to handle multiple responses for each person. After modifying the code to accept multiple responses, you could run these tests and make sure you haven't affected the ability to store a single response or a series of individual responses.

When you're testing your own classes, the `setUp()` method can make your test methods easier to write. You make one set of instances and attributes in `setUp()` and then use these instances in all your test methods. This is much easier than making a new set of instances and attributes in each test method.

NOTE

When a test case is running, Python prints one character for each unit test as it is completed. A passing test prints a dot, a test that results in an error prints an `E`*, and a test that results in a failed assertion prints an* `F`*. This is why you'll see a different number of dots and characters on the first line of output when you run your test cases. If a test case takes a long time to run because it contains many unit tests, you can watch these results to get a sense of how many tests are passing.*

TRY IT YOURSELF

11-3. Employee: Write a class called `Employee`. The `__init__()` method should take in a first name, a last name, and an annual salary, and store each of these as attributes. Write a method called `give_raise()` that adds $5,000 to the annual salary by default but also accepts a different raise amount.

Write a test case for `Employee`. Write two test methods, `test_give_default_raise()` and `test_give_custom_raise()`. Use the `setUp()` method so you don't have to create a new employee instance in each test method. Run your test case, and make sure both tests pass.

Summary

In this chapter you learned to write tests for functions and classes using tools in the `unittest` module. You learned to write a class that inherits from `unittest.TestCase`, and you learned to write test methods that verify specific behaviors your functions and classes should exhibit. You learned to use the `setUp()` method to efficiently create instances and attributes from your classes that can be used in all the test methods for a class.

Testing is an important topic that many beginners don't learn. You don't have to write tests for all the simple projects you try as a beginner. But as soon as you start to work on projects that involve significant development effort, you should test the critical behaviors of your functions and classes. You'll be more confident that new work on your project won't break the parts that work, and this will give you the freedom to make improvements to your code. If you accidentally break existing functionality, you'll know right away, so you can still fix the problem easily. Responding to a failed test that you ran is much easier than responding to a bug report from an unhappy user.

Other programmers respect your projects more if you include some initial tests. They'll feel more comfortable experimenting with your code and be more willing to work with you on projects. If you want to contribute to a project that other programmers are working on, you'll be expected to show that your code passes existing tests and you'll usually be expected to write tests for new behavior you introduce to the project.

Play around with tests to become familiar with the process of testing your code. Write tests for the most critical behaviors of your functions

and classes, but don’t aim for full coverage in early projects unless you have a specific reason to do so.