第11章 测试代码

编写函数或类时，还可为其编写测试。 通过测试，可确定代码面对各种输入都能够按要求的那样工作。 测试让你深信，即便有更多人使用你的程序，它也能正确地工作。

在程序中添加新代码时，也可以对其进行测试，确认不会破坏程序既有的行为。 程序员都会犯错，因此每个程序员都必须经常测试其代码，在用户发现问题前找出它们。

在本章中，将学习如何使用Python模块unittest中的工具来测试代码，还将学习编写测试用例，核实一系列输入都将得到预期的输出。 将看到测试通过了是什么样子，测试未通过又是什么样子，还将知道测试未通过如何有助于改进代码。 将学习如何测试函数和类，并将知道该为项目编写多少个测试。

Murphy's Law (墨菲定律)

if anything can go wrong, it will.

测试驱动开发

英文全称Test-Driven Development，简称TDD，是一种不同于传统软件开发流程的新型的开发方法。它要求在编写某个功能的代码之前先编写测试代码，然后只编写使测试通过的功能代码，通过测试来推动整个开发的进行。这有助于编写简洁可用和高质量的代码，并加速开发过程。

11.1 测试函数

要学习测试，必须有要测试的代码。 下面是一个简单的函数，它接受名和姓并返回整洁的姓名：

In [3]:
```
def get_formatted_name(first, last):
    """生成整洁的姓名。"""
    full_name = f"{first} {last}".title()
    return full_name
```

In [4]:
```
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}.")
```

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

Please give me a first name: Tom
Please give me a last name: Cruise
        Neatly formatted name: Tom Cruise.

Please give me a first name: q
```

从上述输出可知，合并得到的姓名正确无误。 现在假设要修改get_formatted_name()，使其还能够处理中间名。 这样做时，要确保不破坏这个函数处理只含有名和姓的姓名的方式。

为此，可在每次修改get_formatted_name()后都进行测试： 运行程序输入像Janis Joplin这样的姓名。 不过这太烦琐了。 所幸Python提供了一种自动测试函数输出的高效方式。 倘若对get_formatted_name()进行自动测试，就能始终确信当提供测试过的姓名时，该函数都能正确工作。

11.1.1 单元测试和测试用例

Python标准库中的模块unittest提供了代码测试工具。 单元测试用于核实函数的某个方面没有问题。 测试用例是一组单元测试，它们核实函数在各种情形下的行为都符合要求。 良好的测试用例考虑到了函数可能收到的各种输入，包含针对所有这些情形的测试。 全覆盖的测试用例包含一整套单元测试，涵盖了各种可能的函数使用方式。对于大型项目，要进行全覆盖测试可能很难。 通常，最初只要针对代码的重要行为编写测试即可，等项目被广泛使用时再考虑全覆盖。

11.1.2 可通过的测试

你需要一段时间才能习惯创建测试用例的语法，但创建测试用例之后，再添加针对函数的单元测试就很简单了。 要为函数编写测试用例，可先导入模块unittest和要测试的函数，再创建一个继承unittest.TestCase的类，并编写一系列方法对函数行为的不同方面进行测试。

下面的测试用例只包含一个方法，它检查函数get_formatted_name()在给定名和姓时能否正确工作：

```
import unittest
from name_function import get_formatted_name

class NamesTestCase(unittest.TestCase):
    """测试name_function.py。"""

    def test_first_last_name(self):
        """能够正确地处理像Janis Joplin这样的姓名吗？"""
        formatted_name = get_formatted_name('janis', 'joplin')
        self.assertEqual(formatted_name, 'Janis Joplin')

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

我们将直接运行这个文件，但需要指出的是，很多测试框架都会先导入测试文件再运行。 导入文件时，解释器将在导入的同时执行它。if代码块检查特殊变量__name ，**这个变量是在程序执行时设置的。 如果这个文件作为主程序执行，变量__name将被设置为'__main'。 在这里，调用unittest.main()来运行测试用例。 如果这个文件被测试框架导入，变量__name** 的值将不是'__main__'，因此不会调用unittest.main()。

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

第一行的句点表明有一个测试通过了。 接下来的一行指出Python运行了一个测试，消耗的时间不到0.001秒。最后的OK 表明该测试用例中的所有单元测试都通过了。

11.1.3 未通过的测试

测试未通过时结果是什么样的呢？ 我们来修改get_formatted_name()，使其能够处理中间名，但同时故意让该函数无法正确处理像Janis Joplin这样只有名和姓的姓名。

里面包含很多信息，因为测试未通过时，需要让你知道的事情可能有很多。 第一行输出只有一个字母E，指出测试用例中有一个单元测试导致了错误。 接下来，我们看到NamesTestCase中的test_first_last_name()导致了错误。 测试用例包含众多单元测试时，知道哪个测试未通过至关重要。 我们看到了一个标准的traceback，指出函数调用get_formatted_name('janis', 'joplin')有问题，因为缺少一个必不可少的位置实参。 我们还看到运行了一个单元测试。 最后是一条消息，指出整个测试用例未通过，因为运行该测试用例时发生了一个错误。 这条消息位于输出末尾，让你一眼就能看到。 你可不希望为获悉有多少测试未通过而翻阅长长的输出。

In [2]:
```
import unittest
from name_function import get_formatted_name

class NamesTestCase(unittest.TestCase):
    """测试name_function.py。"""

    def test_first_last_name(self):
        """能够正确地处理像Janis Joplin这样的姓名吗？"""
        formatted_name = get_formatted_name('janis', 'joplin')
        self.assertEqual(formatted_name, 'Janis Bob Joplin')

    def test_first_last_name2(self):
        """能够正确地处理像Janis Joplin这样的姓名吗？"""
        formatted_name = get_formatted_name('janis', 'joplin')
        self.assertEqual(formatted_name, 'Janis Bob Joplin')

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

```
E
======================================================================
ERROR: C:\Users\dian\AppData\Roaming\jupyter\runtime\kernel-b400f759-0f94-4c1d-afc5-7b0f
fe2172dc (unittest.loader._FailedTest.C:\Users\dian\AppData\Roaming\jupyter\runtime\kern
el-b400f759-0f94-4c1d-afc5-7b0ffe2172dc)
----------------------------------------------------------------------
AttributeError: module '__main__' has no attribute 'C:\Users\dian\AppData\Roaming\jupyte
r\runtime\kernel-b400f759-0f94-4c1d-afc5-7b0ffe2172dc'

----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (errors=1)
An exception has occurred, use %tb to see the full traceback.

SystemExit: True
C:\Users\dian\anaconda3\Lib\site-packages\IPython\core\interactiveshell.py:3534: UserWar
ning: To exit: use 'exit', 'quit', or Ctrl-D.
  warn("To exit: use 'exit', 'quit', or Ctrl-D.", stacklevel=1)
```

11.1.4 测试未通过时怎么办

测试未通过时怎么办呢？如果你检查的条件没错，测试通过意味着函数的行为是对的，而测试未通过意味着编写的新代码有错。因此，测试未通过时，不要修改测试，而应修复导致测试不能通过的代码： 检查刚刚对函数所做的修改，找出导致函数行为不符合预期的修改。

在本例中，get_formatted_name()以前只需要名和姓两个实参，但现在要求提供名、中间名和姓。 新增的中间名参数是必不可少的，这导致get_formatted_name()的行为不符合预期。 就这里而言，最佳的选择是让中间名变为可选的。 这样做后，使用类似于Janis Joplin的姓名进行测试时，测试就又能通过了，而且也可以接

受中间名。 下面来修改get_formatted_name() ，将中间名设置为可选的，然后再次运行这个测试用例。 如果通过了，就接着确认该函数能够妥善地处理中间名。 要将中间名设置为可选的，可在函数定义中将形参middle移到形参列表末尾，并将其默认值指定为一个空字符串。 还需要添加一个if测试，以便根据是否提供了中间名相应地创建姓名：

11.1.5 添加新测试

确定get_formatted_name()又能正确处理简单的姓名后，我们再编写一个测试，用于测试包含中间名的姓名。为此，在NamesTestCase类中再添加一个方法：

In [6]:
```
class NamesTestCase(unittest.TestCase):
    """测试name_function.py。"""
    def test_first_last_name(self):
        """能够正确地处理像Janis Joplin这样的姓名吗？"""
        formatted_name = get_formatted_name('janis', 'joplin')
        self.assertEqual(formatted_name, 'Janis Joplin')

    def test_first_last_middle_name(self):
        """能够正确地处理像Wolfgang Amadeus Mozart这样的姓名吗？"""
        formatted_name = get_formatted_name('wolfgang', 'mozart', 'amadeus')
        self.assertEqual(formatted_name, 'Wolfgang Amadeus Mozart')
        self.assert
```

11.2 测试类

在本章前半部分，编写了针对单个函数的测试，下面来编写针对类的测试。 很多程序中都会用到类，因此证明你的类能够正确工作大有裨益。 如果针对类的测试通过了，你就能确信对类所做的改进没有意外地破坏其原有的行为。

11.2.1 各种断言方法

Python在unittest.TestCase 类中提供了很多断言方法。 前面说过，断言方法检查你认为应该满足的条件是否确实满足。 如果该条件确实满足，你对程序行为的假设就得到了确认，可以确信其中没有错误。 如果你认为应该满足的条件实际上并不满足，Python将引发异常。

表11-1描述了6个常用的断言方法。 使用这些方法可核实返回的值等于或不等于预期的值，返回的值为True或False，以及返回的值在列表中或不在列表中。 只能在继承unittest.TestCase的类中使用这些方法，随后来看看如何在测试类时使用其中之一。

方法	用途
assertEqual(a, b)	核实a == b
assertNotEqual(a, b)	核实a != b
assertTrue(x)	核实x为True
assertFalse(x)	核实x为False
assertIn(item , list)	核实item在list中
assertNotIn(item , list)	核实item不在list中

11.2.2 一个要测试的类

类的测试与函数的测试相似，所做的大部分工作是测试类中方法的行为。 不过还是存在一些不同之处，下面编写一个要测试的类。 来看一个帮助管理匿名调查的类：

In [2]:
```
class AnonymousSurvey:
    """收集匿名调查问卷的答案。"""
    def __init__(self, question):
        """存储一个问题，并为存储答案做准备。"""
        self.question = question
        self.responses = []
    def show_question(self):
        """显示调查问卷。"""
        print(self.question)
    def store_response(self, new_response):
        """存储单份调查答卷。"""
        self.responses.append(new_response)
    def show_results(self):
        """显示收集到的所有答卷。"""
        print("Survey results:")
        for response in self.responses:
            print(f"- {response}")
```

这个类首先存储了一个调查问题，并创建了一个空列表，用于存储答案。 这个类包含打印调查问题的方法，在答案列表中添加新答案的方法，以及将存储在列表中的答案都打印出来的方法。 要创建该类的实例，只需提供一个问题即可。 有了表示调查的实例后，就可使用show_question()来显示其中的问题，使用store_response()来存储答案并使用show_results()来显示调查结果。 为证明AnonymousSurvey 类能够正确工作，编写一个使用它的程 序：

In [4]:
```
# 定义一个问题，并创建一个调查。
question = "What language did you first learn to program?"
my_survey = AnonymousSurvey(question)
# 显示问题并存储答案。
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)
    # 显示调查结果。
    print("\nThank you to everyone who participated in the survey!")
    my_survey.show_results()
```

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

Language: c

Thank you to everyone who participated in the survey!
Survey results:
- c
Language: c

Thank you to everyone who participated in the survey!
Survey results:
- c
- c
Language: java

Thank you to everyone who participated in the survey!
Survey results:
- c
- c
```

```
- java
Language: python

Thank you to everyone who participated in the survey!
Survey results:
- c
- c
- java
- python
Language: q
```

AnonymousSurvey 类可用于进行简单的匿名调查。假设我们将它放在了模块survey 中，并想进行改进：让每位用户都可输入多个 答案；编写一个方法，只列出不同的答案并指出每个答案出现了多少次；再编写一个类，用于管理非匿名调查。 进行上述修改存在风险，可能影响AnonymousSurvey 类的当前行为。例如，允许每位用户输入多个答案时，可能会不小心修改处理单个答案的方式。要确认在开发这个模块时没有破坏既有行为，可以编写针对这个类的测试。

测试AnonymousSurvey 类

```
import unittest
from survey import AnonymousSurvey


class TestAnonymousSurvey(unittest.TestCase):
    """针对AnonymousSurvey类的测试。"""
    def test_store_single_response(self):
        """测试单个答案会被妥善地存储。"""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
        my_survey.store_response('English')
        self.assertIn('English', my_survey.responses)


if name == 'main':
    unittest.main()
</code>
```

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

OK
```

这很好，但只能收集一个答案的调查用途不大。下面来核实当用户供三个答案时，它们也将被妥善地存储。为此， 在TestAnonymousSurvey 中再添加一个方法：

```
import unittest
from survey import AnonymousSurvey
```

```
class TestAnonymousSurvey(unittest.TestCase):
    """针对AnonymousSurvey类的测试。"""
    def test_store_single_response(self):
        """测试单个答案会被妥善地存储。"""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
        my_survey.store_response('English')
        self.assertIn('English', my_survey.responses)



def test_store_three_responses(self):
    """测试三个答案会被妥善地存储。"""
    question = "What language did you first learn to speak?"
    my_survey = AnonymousSurvey(question)
    responses = ['English', 'Spanish', 'Mandarin']
    for response in responses:
        my_survey.store_response(response)
    for response in responses:
        self.assertIn(response, my_survey.responses)

if name == 'main':
    unittest.main()
</code>
```

我们将该方法命名为test_store_three_responses()，并像对test_store_single_response()所做的一样，在其中创建一个调查对象。定义一个包含三个不同答案的列表，再对其中每个答案调用store_response()。存储这些答案后，使用一个循环来确认每个答案都包含在my_survey.responses中。

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

前述做法的效果很好，但这些测试有些重复的地方。下面使用unittest的另一项功能来高其效率。

在前面的test_survey.py中，我们在每个测试方法中都创建了一个AnonymousSurvey实例，并在每个方法中都创建了答 案。unittest.TestCase类包含的方法setUp()让我们只需创建这些对象一次，就能在每个测试方法中使用。如果在TestCase类中包含了方法setUp()，Python将先运行它，再运行各个以test_打头的方法。这样，在你编写的每个测试方法中，都可使用在方法setUp()中创建的对象。

In [ ]:
下面使用`setUp()`来创建一个调查对象和一组答案，供方法`test_store_single_response()`和`test_store_three_responses()`使用：

```
import unittest
from survey import AnonymousSurvey


class TestAnonymousSurvey(unittest.TestCase):
    """针对AnonymousSurvey类的测试。"""
```

```
    def setUp(self):
        """
        创建一个调查对象和一组答案，供使用的测试方法使用。
        """
        question="What language did you first learn to speak?"
        self.my_survey=AnonymousSurvey(question)
        self.responses=['English','Spanish','Mandarin']



def test_store_single_response(self):
    """测试单个答案会被妥善地存储。"""
    question = "What language did you first learn to speak?"
    my_survey = AnonymousSurvey(question)
    my_survey.store_response('English')
    self.assertIn('English', my_survey.responses)

def test_store_three_responses(self):
    """测试三个答案会被妥善地存储。"""
    question = "What language did you first learn to speak?"
    my_survey = AnonymousSurvey(question)
    responses = ['English', 'Spanish', 'Mandarin']
    for response in responses:
        my_survey.store_response(response)
    for response in responses:
        self.assertIn(response, my_survey.responses)

if name == 'main':
    unittest.main()
</code>
```

方法setUp()做了两件事情：创建一个调查对象，以及创建一个答案列表。存储这两样东西的变量名包含前缀self （即存储在属性中），因此可在这个类的任何地方使用。这让两个测试方法都更简单，因为它们都不用创建调查对象和答案了。方法test_store_single_response()核实self.responses中的第一个答案self.responses[0]被妥善地存储，而方法test_store_three_response()核实self.responses中的全部三个答案都被妥善地存储。

再次运行test_survey.py时，这两个测试也都通过了。如果要扩 展AnonymousSurvey，使其允许每位用户输入多个答案，这些测试将很有用。修改代码以接受多个答案后，可运行这些测试，确认存储单个答案或一系列答案的行为未受影响。

测试自己编写的类时，方法setUp()让测试方法编写起来更容易：可在setUp()方法中创建一系列实例并设置其属性，再在测试方法中直接使用这些实例。相比于在每个测试方法中都创建实例并设置其属性，这要容易得多。

注意　运行测试用例时，每完成一个单元测试，Python都打印一个字符：测试通过时打印一个句点，测试引发错误时打印一个E，而测试导致断言失败时则打印一个F。这就是你运行测试用例时，在输出的第一行中看到的句点和字符数量各不相同的原因。如果测试用例包含很多单元测试，需要运行很长时 间，就可通过观察这些结果来获悉有多少个测试通过了。

小结

在本章中，你学习了：如何使用模块unittest中的工具来为函数和类编写测试；如何编写继承unittest.TestCase的类，以及如何编写测试方法，以核实函数和类的行为符合预期；如何使用方法setUp()来根据类高效地创建实例并设置其属性，以便在类的所有测试方法中使用。

In [ ]: