

30 | 真的有必要写单元测试吗？

2019-07-17 景霄

Python核心技术与实战

[进入课程 >](#)



讲述：冯永吉

时长 10:43 大小 9.83M



你好，我是景霄。

说到 unit test（即单元测试，下文统一用中文称呼），大部分人的反应估计有这么两种：要么就是，单元测试啊，挺简单的呀，做不做无所谓吧；要么就是，哎呀，项目进度太赶，单元测试拖一拖之后再来吧。

显然，这两种人，都没有正确认识到单元测试的价值，也没能掌握正确的单元测试方法。你是不是觉得自己只要了解 Python 的各个 feature，能够编写出符合规定功能的程序就可以了呢？

其实不然，完成产品的功能需求只是很基础的一部分，如何保证所写代码的稳定、高效、无误，才是我们工作的关键。而学会合理地使用单元测试，正是帮助你实现这一目标的重要路

径。


我们总说，测试驱动开发（TDD）。今天我就以 Python 为例，教你设计编写 Python 的单元测试代码，带你熟悉并掌握这一重要技能。

什么是单元测试？

单元测试，通俗易懂地讲，就是编写测试来验证某一个模块的功能正确性，一般会指定输入，验证输出是否符合预期。

实际生产环境中，我们会对每一个模块的所有可能输入值进行测试。这样虽然显得繁琐，增加了额外的工作量，但是能够大大提高代码质量，减小 bug 发生的可能性，也更方便系统的维护。

说起单元测试，就不得不提 [Python unittest 库](#)，它提供了我们需要的大多数工具。我们来看下面这个简单的测试，从代码中了解其使用方法：

 复制代码


```
1 import unittest
2
3 # 将要被测试的排序函数
4 def sort(arr):
5     l = len(arr)
6     for i in range(0, l):
7         for j in range(i + 1, l):
8             if arr[i] >= arr[j]:
9                 tmp = arr[i]
10                arr[i] = arr[j]
11                arr[j] = tmp
12
13
14 # 编写子类继承 unittest.TestCase
15 class TestSort(unittest.TestCase):
16
17     # 以 test 开头的函数将会被测试
18     def test_sort(self):
19         arr = [3, 4, 1, 5, 6]
20         sort(arr)
21         # assert 结果跟我们期待的一样
22         self.assertEqual(arr, [1, 3, 4, 5, 6])
23
24 if __name__ == '__main__':
25     ## 如果在 Jupyter 下，请用如下方式运行单元测试
26     unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

```
27
28     ## 如果是命令行下运行，则：
29     ## unittest.main()
30
31 ## 输出
32 ..
33 -----
34 Ran 2 tests in 0.002s
35
36 OK
```

这里，我们创建了一个排序函数的单元测试，来验证排序函数的功能是否正确。代码里我做了非常详细的注释，相信你能够大致读懂，我再来介绍一些细节。

首先，我们需要创建一个类 `TestSort`，继承类 `'unittest.TestCase'`；然后，在这个类中定义相应的测试函数 `test_sort()`，进行测试。注意，测试函数要以 `'test'` 开头，而测试函数的内部，通常使用 `assertEqual()`、`assertTrue()`、`assertFalse()` 和 `assertRaise()` 等 `assert` 语句对结果进行验证。

最后运行时，如果你是在 IPython 或者 Jupyter 环境下，请使用下面这行代码：

 复制代码

```
1 unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

而如果你用的是命令行，直接使用 `unittest.main()` 就可以了。你可以看到，运行结果输出 `'OK'`，这就表示我们的测试通过了。

当然，这个例子中的被测函数相对简单一些，所以写起对应的单元测试来也非常自然，并不需要很多单元测试的技巧。但实战中的函数往往还是比较复杂的，遇到复杂问题，高手和新手的最大差别，便是单元测试技巧的使用。

单元测试的几个技巧


接下来，我将会介绍 Python 单元测试的几个技巧，分别是 `mock`、`side_effect` 和 `patch`。这三者用法不一样，但都是一个核心思想，即用**虚假的实现，来替换掉被测试函数的一些依赖项**，让我们能把更多的精力放在需要被测试的功能上。

mock

mock 是单元测试中最核心重要的一环。mock 的意思，便是通过一个虚假对象，来代替被测试函数或模块需要的对象。

举个例子，比如你要测一个后端 API 逻辑的功能性，但一般后端 API 都依赖于数据库、文件系统、网络等。这样，你就需要通过 mock，来创建一些虚假的数据库层、文件系统层、网络层对象，以便可以简单地对核心后端逻辑单元进行测试。

Python mock 则主要使用 mock 或者 MagicMock 对象，这里我也举了一个代码示例。这个例子看上去比较简单，但是里面的思想很重要。下面我们一起来看下：

 复制代码

```
1 import unittest
2 from unittest.mock import MagicMock
3
4 class A(unittest.TestCase):
5     def m1(self):
6         val = self.m2()
7         self.m3(val)
8
9     def m2(self):
10         pass
11
12     def m3(self, val):
13         pass
14
15     def test_m1(self):
16         a = A()
17         a.m2 = MagicMock(return_value="custom_val")
18         a.m3 = MagicMock()
19         a.m1()
20         self.assertTrue(a.m2.called) # 验证 m2 被 call 过
21         a.m3.assert_called_with("custom_val") # 验证 m3 被指定参数 call 过
22
23 if __name__ == '__main__':
24     unittest.main(argv=['first-arg-is-ignored'], exit=False)
25
26 ## 输出
27 ..
28 -----
29 Ran 2 tests in 0.002s
30
31 OK
```

这段代码中，我们定义了一个类的三个方法 m1()、m2()、m3()。我们需要对 m1() 进行单元测试，但是 m1() 取决于 m2() 和 m3()。如果 m2() 和 m3() 的内部比较复杂，你就不能只是简单地调用 m1() 函数来进行测试，可能需要解决很多依赖项的问题。

这一听就让人头大了吧？但是，有了 mock 其实就很好办了。我们可以把 m2() 替换为一个返回具体数值的 value，把 m3() 替换为另一个 mock（空函数）。这样，测试 m1() 就很容易了，我们可以测试 m1() 调用 m2()，并且用 m2() 的返回值调用 m3()。


可能你会疑惑，这样测试 m1() 不是基本上毫无意义吗？看起来只是象征性地测了一下逻辑呀？

其实不然，真正工业化的代码，都是很多层模块相互逻辑调用的一个树形结构。单元测试需要测的是某个节点的逻辑功能，mock 掉相关的依赖项是非常重要的。这也是为什么会被叫做单元测试 unit test，而不是其他的 integration test、end to end test 这类。

Mock Side Effect

第二个我们来看 Mock Side Effect，这个概念很好理解，就是 mock 的函数，属性是可以根据不同的输入，返回不同的数值，而不只是一个 return_value。


比如下面这个示例，例子很简单，测试的是输入参数是否为负数，输入小于 0 则输出为 1，否则输出为 2。代码很简短，你一定可以看懂，这便是 Mock Side Effect 的用法。

 复制代码

```
1 from unittest.mock import MagicMock
2 def side_effect(arg):
3     if arg < 0:
4         return 1
5     else:
6         return 2
7 mock = MagicMock()
8 mock.side_effect = side_effect
9
10 mock(-1)
11 1
12
13 mock(1)
14 2
```

patch


至于 patch，给开发者提供了非常便利的函数 mock 方法。它可以应用 Python 的 decoration 模式或是 context manager 概念，快速自然地 mock 所需的函数。它的用法也不难，我们来看代码：

 复制代码

```
1 from unittest.mock import patch
2
3 @patch('sort')
4 def test_sort(self, mock_sort):
5     ...
6     ...
```

在这个 test 里面，mock_sort 替代 sort 函数本身的存在，所以，我们可以像开始提到的 mock object 一样，设置 return_value 和 side_effect。

另一种 patch 的常见用法，是 mock 类的成员函数，这个技巧我们在工作中也经常用到，比如说一个类的构造函数非常复杂，而测试其中一个成员函数并不依赖所有初始化的 object。它的用法如下：

 复制代码

```
1 with patch.object(A, '__init__', lambda x: None):
2     ...
```

代码应该也比较易懂。在 with 语句里面，我们通过 patch，将 A 类的构造函数 mock 为一个 do nothing 的函数，这样就可以很方便地避免一些复杂的初始化（initialization）。

其实，综合前面讲的这几点来看，你应该感受到了，单元测试的核心还是 mock，mock 掉依赖项，测试相应的逻辑或算法的准确性。在我看来，虽然 Python unittest 库还有很多层出不穷的方法，但只要你能掌握了 MagicMock 和 patch，编写绝大部分工作场景的单元测试就不成问题了。

高质量单元测试的关键

这节课的最后，我想谈一谈高质量的单元测试。我很理解，单元测试这个东西，哪怕是正在使用的人也是“百般讨厌”的，不少人很多时候只是敷衍了事。我也嫌麻烦，但从来不敢松懈，因为在大公司里，如果你写一个很重要的模块功能，不写单元测试是无法通过 code review 的。

低质量的单元测试，可能真的就是摆设，根本不能帮我们验证代码的正确性，还浪费时间。那么，既然要做单元测试，与其浪费时间糊弄自己，不如追求高质量的单元测试，切实提高代码品质。

那该怎么做呢？结合工作经验，我认为一个高质量的单元测试，应该特别关注下面两点。

Test Coverage

首先我们要关注 Test Coverage，它是衡量代码中语句被 cover 的百分比。可以说，提高代码模块的 Test Coverage，基本等同于提高代码的正确性。

为什么呢？

要知道，大多数公司代码库的模块都非常复杂。尽管它们遵从模块化设计的理念，但因为有复杂的业务逻辑在，还是会产生逻辑越来越复杂的模块。所以，编写高质量的单元测试，需要我们 cover 模块的每条语句，提高 Test Coverage。

我们可以用 Python 的 coverage tool 来衡量 Test Coverage，并且显示每个模块为被 coverage 的语句。如果你想了解更多更详细的使用，可以点击这个链接来学习：

<https://coverage.readthedocs.io/en/v4.5.x/>。

模块化

高质量单元测试，不仅要求我们提高 Test Coverage，尽量让所写的测试能够 cover 每个模块中的每条语句；还要求我们从测试的角度审视 codebase，去思考怎么模块化代码，以便写出高质量的单元测试。

光讲这段话可能有些抽象，我们来看这样的场景。比如，我写了一个下面这个函数，对一个数组进行处理，并返回新的数组：

```


1 def work(arr):
2     # pre process
3     ...
4     ...
5     # sort
6     l = len(arr)
7     for i in range(0, l):
8         for j in range(i + 1, j):
9             if arr[i] >= arr[j]:
10                tmp = arr[i]
11                arr[i] = arr[j]
12                arr[j] = tmp
13     # post process
14     ...
15     ...
16     Return arr

```

这段代码的大概意思是，先有个预处理，再排序，最后再处理一下然后返回。如果现在要求你，给这个函数写个单元测试，你是不是会一筹莫展呢？

毕竟，这个函数确实有点儿复杂，以至于你都不知道应该是什么样的输入，并要期望怎样的输出。这种代码写单元测试是非常痛苦的，更别谈 cover 每条语句的要求了。

所以，正确的测试方法，应该是先模块化代码，写成下面的形式：


 复制代码

```

1 def preprocess(arr):
2     ...
3     ...
4     return arr
5
6 def sort(arr):
7     ...
8     ...
9     return arr
10
11 def postprocess(arr):
12     ...
13     return arr
14
15 def work(self):
16     arr = preprocess(arr)
17     arr = sort(arr)
18     arr = postprocess(arr)
19     return arr

```


接着再进行相应的测试，测试三个子函数的功能正确性；然后通过 mock 子函数，调用 work() 函数，来验证三个子函数被 call 过。

 复制代码

```
1 from unittest.mock import patch
2
3 def test_preprocess(self):
4     ...
5
6 def test_sort(self):
7     ...
8
9 def test_postprocess(self):
10    ...
11
12 @patch('%s.preprocess')
13 @patch('%s.sort')
14 @patch('%s.postprocess')
15 def test_work(self, mock_post_process, mock_sort, mock_preprocess):
16     work()
17     self.assertTrue(mock_post_process.called)
18     self.assertTrue(mock_sort.called)
19     self.assertTrue(mock_preprocess.called)
```

你看，这样一来，通过重构代码就可以使单元测试更加全面、精确，并且让整体架构、函数设计都美观了不少。

总结

回顾下这节课，整体来看，单元测试的理念是先模块化代码设计，然后针对每个作用单元，编写单独的测试去验证其准确性。更好的模块化设计和更多的 Test Coverage，是提高代码质量的核心。而单元测试的本质就是通过 mock，去除掉不影响测试的依赖项，把重点放在需要测试的代码核心逻辑上。

讲了这么多，还是想告诉你，单元测试是个非常非常重要的技能，在实际工作中是保证代码质量和准确性必不可少的一环。同时，单元测试的设计技能，不只是适用于 Python，而是适用于任何语言。所以，单元测试必不可少。

思考题

那么，你在平时的学习工作中，曾经写过单元测试吗？在编写单元测试时，用到过哪些技巧或者遇到过哪些问题吗？欢迎留言与我交流，也欢迎你把这篇文章分享出去。



Python 核心技术与实战

系统提升你的 Python 能力

景霄

Facebook 资深工程师



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 29 | 巧用上下文管理器和With语句精简代码

下一篇 31 | pdb & cProfile：调试和性能分析的法宝

精选留言 (10)

写留言



KaitoShy

2019-07-17

后面有几个代码没怎么看懂，希望老师详细说明一下：

```
1) from unittest.mock import patch
```

```
@patch('sort')
```

```
def test_sort(self, mock_sort):...
```

展开 ▾

3

3



□

2019-07-17

我的单元测试是直接跑流程，有时候有些很好复杂的调用就很麻烦。这个应该能提高效率。

1

1



小侠龙旋风

2019-07-21

在说Mock Side Effect的用法时，老师，你说代码很简短，你一定可以看懂。然后我就对这两句疑惑了：

```
mock = MagicMock()
```

```
mock.side_effect = side_effect
```

在mock.side_effect = side_effect赋值中到底发生了什么？

展开 ▾

1

1



assert

2019-07-18

受益匪浅

展开 ▾

1

1



响雨

2019-07-18

单元测试中current_app这样的上下文环境，怎么导入啊

1

1



王校

2019-07-18

这节让我一下子对单元测试理解了不少。我还想进一步了解 更多技巧和 最佳实践。有什么好的学习链接和书籍推荐吗？

展开 ▾



夜路破晓

2019-07-17

认知层次决定了效率高低。

虽然作为小白代码部分看得一脸懵逼，但完全get到了测试单元属于高级思维运用的高级方法与技能，因为它不仅要求代码设计拥有模块化理念的底层逻辑，还提倡代码不只是满足产品功能需求更要求持续稳定高效。

这就是码农与非码农的认知差距。

展开 ∨



Claywoow

2019-07-17

一些大项目中的函数有文件的读写操作有必要mock掉吗

作者回复: 最好也mock一下



enjoylearning

2019-07-17

很喜欢写单测，通过后才会继续实现下一步功能

展开 ∨

作者回复: 嗯嗯，这个习惯很好



return

2019-07-17

一直因为业务逻辑复杂，而不好做单元测试。今天茅塞顿开。感谢老师。

作者回复: 谢谢你的支持



