关于作者

廖雪峰 北京 朝阳区

+ 加关注

单元测试

如果你听说过“测试驱动开发”（TDD：Test-Driven Development），单元测试就不陌生。

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

比如对函数`abs()`，我们可以编写出以下几个测试用例：

1. 输入正数，比如`1`、`1.2`、`0.99`，期待返回值与输入相同；
2. 输入负数，比如`-1`、`-1.2`、`-0.99`，期待返回值与输入相反；
3. 输入`0`，期待返回`0`；
4. 输入非数值类型，比如`None`、`[]`、`{}`，期待抛出`TypeError`。

把上面的测试用例放到一个测试模块里，就是一个完整的单元测试。

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，总之，需要修复使单元测试能够通过。

单元测试通过后有什么意义呢？如果我们对`abs()`函数代码做了修改，只需要再跑一遍单元测试，如果通过，说明我们的修改不会对`abs()`函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在将来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

我们来编写一个`Dict`类，这个类的行为和`dict`一致，但是可以通过属性来访问，用起来就像下面这样：

```
>>> d = Dict(a=1, b=2)
>>> d['a']
1
>>> d.a
1
```

`mydict.py`代码如下：

```
class Dict(dict):

    def __init__(self, **kw):
        super().__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value
```

为了编写单元测试，我们需要引入Python自带的`unittest`模块，编写`mydict_test.py`如下：

```
import unittest

from mydict import Dict

class TestDict(unittest.TestCase):

    def test_init(self):
        d = Dict(a=1, b='test')
        self.assertEqual(d.a, 1)
        self.assertEqual(d.b, 'test')
        self.assertTrue(isinstance(d, dict))

    def test_key(self):
        d = Dict()
        d['key'] = 'value'
        self.assertEqual(d.key, 'value')

    def test_attr(self):
        d = Dict()
        d.key = 'value'
        self.assertTrue('key' in d)
        self.assertEqual(d['key'], 'value')

    def test_keyerror(self):
        d = Dict()
        with self.assertRaises(KeyError):
            value = d['empty']

    def test_attrerror(self):
        d = Dict()
        with self.assertRaises(AttributeError):
            value = d.empty
```

编写单元测试时，我们需要编写一个测试类，从`unittest.TestCase`继承。

以`test`开头的方法就是测试方法，不以`test`开头的方法不被认为是测试方法，测试的时候不会被执行。

对每一类测试都需要编写一个`test_xxx()`方法。由于`unittest.TestCase`提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是`assertEqual()`：

```
self.assertEqual(abs(-1), 1) # 断言函数返回的结果与1相等
```

另一种重要的断言就是期待抛出指定类型的Error，比如通过`d['empty']`访问不存在的key时，断言会抛出`KeyError`：

```
with self.assertRaises(KeyError):
    value = d['empty']
```

而通过`d.empty`访问不存在的key时，我们期待抛出`AttributeError`：

```
with self.assertRaises(AttributeError):
    value = d.empty
```

运行单元测试

一旦编写好单元测试，我们就可以运行单元测试。最简单的运行方式是在`mydict_test.py`的最后加上两行代码：

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

这样就可以把`mydict_test.py`当做正常的python脚本运行：

```
$ python mydict_test.py
```

另一种方法是在命令行通过参数`-m unittest`直接运行单元测试：

```
$ python -m unittest mydict_test
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

这是推荐的做法，因为这样可以一次批量运行很多单元测试，并且，有很多工具可以自动来运行这些单元测试。

setUp与tearDown

可以在单元测试中编写两个特殊的`setUp()`和`tearDown()`方法。这两个方法会分别在每调用一个测试方法的前后分别被执行。

`setUp()`和`tearDown()`方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在`setUp()`方法中连接数据库，在`tearDown()`方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码：

```
class TestDict(unittest.TestCase):

    def setUp(self):
        print('setUp...')

    def tearDown(self):
        print('tearDown...')
```

可以再次运行测试看看每个测试方法调用前后是否会打印出`setUp...`和`tearDown...`。

练习

对Student类编写单元测试，结果发现测试不通过，请修改Student类，让测试通过：

```
# -*- coding: utf-8 -*-
import unittest

class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def get_grade(self):
        if self.score >= 60:
            return 'B'
        if self.score >= 80:
            return 'A'
        return 'C'
```

```
class TestStudent(unittest.TestCase):

    def test_80_to_100(self):
        s1 = Student('Bart', 80)
        s2 = Student('Lisa', 100)
        self.assertEqual(s1.get_grade(), 'A')
        self.assertEqual(s2.get_grade(), 'A')

    def test_60_to_80(self):
        s1 = Student('Bart', 60)
        s2 = Student('Lisa', 79)
        self.assertEqual(s1.get_grade(), 'B')
        self.assertEqual(s2.get_grade(), 'B')

    def test_0_to_60(self):
        s1 = Student('Bart', 0)
        s2 = Student('Lisa', 59)
        self.assertEqual(s1.get_grade(), 'C')
        self.assertEqual(s2.get_grade(), 'C')

    def test_invalid(self):
        s1 = Student('Bart', -1)
        s2 = Student('Lisa', 101)
        with self.assertRaises(ValueError):
            s1.get_grade()
        with self.assertRaises(ValueError):
            s2.get_grade()

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

Run

小结

单元测试可以有效地测试某个程序模块的行为，是未来重构代码的信心保证。

单元测试的测试用例要覆盖常用的输入组合、边界条件和异常。

单元测试代码要非常简单，如果测试代码太复杂，那么测试代码本身就可能有bug。

单元测试通过了并不意味着程序就没有bug了，但是不通过程序肯定有bug。

参考源码

mydict.py

mydict_test.py

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：

还可以分享给朋友：

分享到微博

Previous Page Next Page

Comments

Make a comment

Sign in to make a comment