学习目标

1. 掌握 TestCase 的使用;
2. 掌握通过 TextTestRunner 执行 TestCase 中的测试用例;
3. 掌握使用 TestLoader 加载 TestCase 中的测试用例;
4. 掌握常用断言 assertIn 和 assertEqual 的使用语法;
5. 了解参数化技术;
6. 掌握生成 HTML 格式的测试报告;

目录

第 1 章 UnitTest..... 3
第 2 章 Fixture..... 9
第 3 章 断言..... 12
第 4 章 参数化..... 14
第 5 章 跳过..... 17
第 6 章 生成 HTML 测试报告..... 19

第 1 章 UnitTest

一、UnitTest 基本使用

说明：

- 框架英文单词 framework；
- 为解决一类事情的功能集合；

1. 什么是 UnitTest 框架？

概念：UnitTest 是 Python 自带的一个单元测试框架，用它来做单元测试。

2. 为什么使用 UnitTest 框架？

- 能够组织多个用例去执行；
- 提供丰富的断言方法；
- 能够生成测试报告；

3. UnitTest 核心要素

- TestCase；
- TestSuite；
- TestRunner；
- TestLoader；
- Fixture；

4. TestCase

说明：TestCase 就是测试用例的意思。

案例：

定义一个实现加法操作的函数，并对该函数进行测试。

定义测试用例：

1、导包：import unittest;

2、定义测试类：新建测试类必须继承 unittest.TestCase;

3、定义测试方法：测试方法名称命名必须以 test 开头;

4、调用 unittest.main() 执行测试用例 ；

```
# a.py
# 导包
import unittest
def my_sum(a, b):
    return a + b

class my_test(unittest.TestCase):
    def test_01(self):
        print(my_sum(4, 6))

    def test_02(self):
        print(my_sum(3, 2))
```

思考：如何同时运行多个 py 文件中的测试用例?

二、TestSuite

说明：(翻译：测试套件)多条测试用例集合在一起，就是一个 TestSuite。

使用：

1、实例化： suite = unittest.TestSuite()

(suite：为 TestSuite 实例化的名称)

2、添加用例：suite.addTest(ClassName("MethodName"))

(ClassName：为类名；MethodName：为方法名)

3、添加扩展：suite.addTest(unittest.makeSuite(ClassName))

(搜索指定 ClassName 内 test 开头的方法并添加到测试套件中）。

提示：TestSuite 需要配合 TextTestRunner 才能被执行

三、TextTestRunner

说明：TextTestRunner 是用来执行测试用例和测试套件的

使用：

1、实例化： runner = unittest.TextTestRunner()

2、执行： runner.run(suite) # suite：为测试套件名称

四、案例：

将 test01.py..test02.py 共 2 条用例，将这 2 条用例批量执行；

1. 示例代码

test01.py

```
# test01.py
# 导包
import unittest
class my_test01(unittest.TestCase):
    def test_01(self):
        print("my_test01 的 test01")

    def test_02(self):
        print("my_test01 的 test02")
```

test02.py

```
# test02.py
# 导包
import unittest
class my_test02(unittest.TestCase):
    def test_01(self):
        print("my_test02 的 test01")

    def test_02(self):
        print("my_test02 的 test02")
```

执行测试用例

```
import unittest
import test01
import test02
suite = unittest.TestSuite()
suite.addTest(test01.my_test01("test_01"))
# 添加 my_test02 类中所有 test 开头的方法
suite.addTest(unittest.makeSuite(test02.my_test02))
runner = unittest.TextTestRunner()
runner.run(suite)
```

问题：使用 suite.addtest(unittest.makeSuite(className))导入多条测试用例，addtest()需要调用多次。

五、TestLoader

说明：用来加载 TestCase 到 TestSuite 中，即加载满足条件的测试用例，并把测试用例封装成测试套件。

使用 unittest.TestLoader，通过该类下面的 discover()方法自动搜索指定目录下指定开头的.py 文件，并将查找到的测试用例组装到测试套件；

用法：suite = unittest.TestLoader().discover(test_dir, pattern='test*.py')

自动搜索指定目录下指定开头的.py 文件，并将查找到的测试用例组装到测试套件；

test_dir：为指定的测试用例的目录；

pattern：为查找的.py 文件的格式；

如果文件名默认为'test*.py' 也可以使用 unittest.defaultTestLoader 代替 unittest.TestLoader()

运行：

```
runner = unittest.TextTestRunner()
runner.run(suite)
```

1. 示例代码

test01.py

```
# test01.py
# 导包
import unittest
class my_test01(unittest.TestCase):
    def test_01(self):
        print("my_test01 的 test01")

    def test_02(self):
        print("my_test01 的 test02")
```

test02.py

```
# test02.py
# 导包
import unittest
class my_test02(unittest.TestCase):
    def test_01(self):
        print("my_test02 的 test01")

    def test_02(self):
        print("my_test02 的 test02")
```

执行测试用例

```
import unittest
suite = unittest.TestLoader().discover("./", "test*.py")
# suite = unittest.defaultTestLoader.discover("./")
runner = unittest.TextTestRunner()
runner.run(suite)
```

六、TestSuite 与 TestLoader 区别

- TestSuite 需要手动添加测试用例（可以添加测试类，也可以添加测试类中某个测试方法）；
- TestLoader 搜索指定目录下指定开头.py 文件，并添加测试类中的所有的测试方法，不能指定添加测试方法；

第 2 章 Fixture

一、Fixture 介绍

说明：Fixture 是一个概述，对一个测试用例环境的初始化和销毁就是一个 Fixture 。

二、Fixture 控制级别：

- 方法级别；
- 类级别；
- 模块级别；

三、方法级别

使用：

- 初始化(前置处理): def setUp(self) --> 首先自动执行；

- 销毁(后置处理): def tearDown(self) --> 最后自动执行;

运行于测试方法的始末，即：运行一次测试方法就会运行一次 setUp 和 tearDown

1. 示例代码

test01.py

```
# test01.py
import unittest
class my_test01(unittest.TestCase):
    def setUp(self):
        print("setUP 执行")

    def tearDown(self):
        print("teardown 执行")

    def test_01(self):
        print("my_test01 的 test01")

    def test_02(self):
        print("my_test01 的 test02")
```

四、类级别

使用:

- 初始化(前置处理): @classmethod def setUpClass(cls): --> 首先自动执行
- 销毁(后置处理): @classmethod def tearDownClass(cls): --> 最后自动执行

运行于测试类的始末，即：每个测试类只会运行一次 setUpClass 和 tearDownClass

1. 示例代码

test01.py

```python
# test01.py
# 导包
import unittest
class my_test01(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        print("setUPClass 执行")

    @classmethod
    def tearDownClass(cls):
        print("tearDownClass 执行")

    def setUp(self):
        print("setUP 执行")

    def tearDown(self):
        print("teardown 执行")

    def test_01(self):
        print("my_test01 的 test01")

    def test_02(self):
        print("my_test01 的 test02")
```

五、模块级别 [了解]

使用:

- 初始化(前置处理): def setUpModule() --> 首先自动执行
- 销毁(后置处理): def tearDownModule() --> 最后自动执行

运行于整个模块的始末，即：整个模块只会运行一次 setUpModule 和

tearDownModule

六、总结

- 必须继承 unittest.TestCase 类，setUp、tearDown 才是一个 Fixture；
- setUp，tearDown：如果一个类中有多个测试用例，每执行一个测试用例之前会调用一次 setUp，之后会调用一次 tearDown；
- setUpClass，tearDownClass：如果一个类中有多个测试用例，执行所有测试用例之前只会调用一次 setUpClass，之后只会调用一次 tearDownClass；
- setUpModule，tearDownModule：只在 import 导入这个模块时会调用一次 setUpModule，模块使用完成之后会调用一次 tearDownModule；
- setUpXXX：一般做初始化工作；
- tearDownXXX：一般做结束工作；

第 3 章 断言

一、目标

- 理解什么是断言；
- 掌握断言 assertEqual、assertIn 方法；
- 了解 UnitTest 其他断言方法。

二、UnitTest 断言

概念：让程序代替人为判断测试程序执行结果是否符合预期结果的过程。

三、为什么要学习断言？

自动化脚本在执行的时候一般都是无人值守状态，我们不知道执行结果是否符合预期结果，所以我们需要让程序代替人为检测程序执行的结果是否符合预期结果，这就需要使用断言。

四、UnitTest 常用断言方法

说明：

UnitTest 中提供了非常丰富的断言方法，请参考附件资料。

复杂的断言方法在自动化测试中几乎使用不到，所以我们只需要掌握几个常用的即可。

常用的 UnitTest 断言方法：

序号	断言方法	断言描述
1	assertTrue(expr, msg=None)	验证 expr 是 true，如果为 false，则 fail
2	assertFalse(expr, msg=None)	验证 expr 是 false，如果为 true，则 fail
3	assertEqual(expected, actual, msg=None)	验证 expected==actual，不等则 fail
4	assertNotEqual(first, second, msg=None)	验证 first != second，相等则 fail
5	assertIsNone(obj, msg=None)	验证 obj 是 None，不是则 fail
6	assertIsNotNone(obj, msg=None)	验证 obj 不是 None，是则 fail
7	assertIn(member, container, msg=None)	验证是否 member in container
8	assertNotIn(member, container, msg=None)	验证是否 member not in container

五、使用方式

断言方法已经在 unittest.TestCase 类中定义好了，而且我们自定义的测试类已

经继承了 TestCase，所以在测试方法中直接调用即可。

```
# a.py
import unittest
def my_sum(a, b):
    return a + b

class my_test(unittest.TestCase):
    def test01(self):
        num = my_sum(1, 3)
        # 如果num为4，正确
        self.assertEqual(4, num)

    def test02(self):
        num = my_sum(4, 3)
        # 如果num为4，正确
        self.assertEqual(4, num)

    def test03(self):
        num = my_sum(1, 2)
        # 如果num在列表中，正确
        self.assertIn(num, [1, 2, 3, 4, 5])
```


第 4 章 参数化

一、目标

掌握如何实现参数化。

二、需求

定义一个实现加法操作的函数，并对该函数进行测试。

1. 示例代码

```
# a.py
import unittest
# 求和
def my_sum(a, b):
    return a + b

class TestAdd(unittest.TestCase):
    def test_01(self):
        result = my_sum(1, 1)
        self.assertEqual(result, 2)

    def test_02(self):
        result = my_sum(1, 0)
        self.assertEqual(result, 1)

    def test_03(self):
        result = my_sum(0, 0)
        self.assertEqual(result, 0)
```

发现问题

- 一条测试数据定义一个测试函数，代码冗余度太高；

三、参数化

通过参数化的方式来传递数据，从而实现数据和脚本分离。并且可以实现用例的重复执行。unittest 测试框架，本身不支持参数化，但是可以通过安装 unittest 扩展插件 parameterized 来实现。

安装

pip install parameterized

使用方式

- 导包：from parameterized import parameterized;
- 使用@parameterized.expand 装饰器可以为测试函数的参数进行参数化;

方式一:

```
# a.py
import unittest
from parameterized import parameterized
# 求和
def my_sum(a, b):
    return a + b

class Testmy_sum(unittest.TestCase):
    @parameterized.expand([(1, 1, 2), (1, 0, 1), (0, 0, 0)])
    def test_01(self, x, y, expect):
        result = my_sum(x, y)
        self.assertEqual(result, expect)
```

方式二:

```
# a.py
import unittest
from parameterized import parameterized
# 求和
def my_sum(a, b):
    return a + b

# 构建测试数据
data = [(1, 1, 2), (1, 0, 1), (0, 0, 0)]

class Testmy_sum(unittest.TestCase):
    @parameterized.expand(data)
    def test_02(self, x, y, expect):
        result = my_sum(x, y)
        self.assertEqual(result, expect)
```

```
```

方式三:

```
# a.py
import unittest
from parameterized import parameterized
# 求和
def my_sum(a, b):
    return a + b

# 构建测试数据
def build_data():
    return [(1, 1, 2), (1, 0, 1), (0, 0, 0)]

class Testmy_sum(unittest.TestCase):
    @parameterized.expand(build_data)
    def test_03(self, x, y, expect):
        result = my_sum(x, y)
        self.assertEqual(result, expect)
```

第 5 章 跳过

一、目标

掌握如何把测试函数标记成跳过。

二、跳过

对于一些未完成的或者不满足测试条件的测试函数和测试类，可以跳过执行。

三、使用方式

```
# 直接将测试函数标记成跳过
@unittest.skip('代码未完成')
# 根据条件判断测试函数是否跳过
@unittest.skipIf(condition, reason)
```

1. 示例代码

```
# a.py
import unittest
version = 35
class My_Test1(unittest.TestCase):
    @unittest.skip("代码未完成")
    def test_01(self):
        print("test_01")

    @unittest.skipIf(version <= 30, "版本大于 30 才会执行")
    def test_02(self):
        print("test_02")

@unittest.skip("代码未完成")
class My_Test2(unittest.TestCase):
    def test_a(self):
        print("test_a")

    def test_b(self):
        print("test_b")
```

第 6 章 生成 HTML 测试报告

一、目标

掌握如何生成 HTML 测试报告方法。

二、什么是 HTML 测试报告

说明：HTML 测试报告就是执行完测试用例后，以 HTML(网页)方式将执行结果生成报告。

三、为什么要生成测试报告

- 测试报告是本次测试结果的体现形态；
- 测试报告内包含了有关本次测试用例的详情；

四、HTML 生成报告方式

- TextTestRunner （UnitTest 自带）；
- HTMLTestRunner（第三方模板）；

五、生成 TextTestRunner 测试报告

1、导入 unittest 包；

2、生成测试套件 suite = unittest.TestLoader().discover("./", "test*.py") ；

3、以只写方式打开测试报告文件 f = open("test01.txt", "w", encoding="utf-8");

4、实例化 HTMLTestRunner 对象： runner = unittest.TextTestRunner(stream=f, verbosity=2)

参数说明： stream 为 open 函数打开的文件流；
verbosity 为不同模板编号

5、执行：runner.run(suite)；

6、关闭文件；

1. 示例代码

```
import unittest
suite = unittest.TestLoader().discover("./", "test*.py")
f = open("test01.txt", "w", encoding="utf-8")
runner = unittest.TextTestRunner(stream=f, verbosity=2)
runner.run(suite)
f.close()
```

六、生成 HTMLTestRunner 测试报告

1、复制 HTMLTestRunner.py 文件到项目文件夹；

2、导入 HTMLTestRunner、unittest 包；

3、生成测试套件 suite = unittest.TestLoader().discover("./", "test*.py") ；

4、以只写方式打开测试报告文件 f = open("test01.html", "wb") ；

5、实例化 HTMLTestRunner 对象： runner = HTMLTestRunner(stream=f, title="自动化测试报告", description="Chrome 浏览器")

参数说明：
stream：open 函数打开的文件流；
title：[可选参数]，为报告标题；
description：[可选参数]，为报告描述信息；比如操作系统、浏览器等版本；

6、执行：runner.run(suite)；

7、关闭文件;

1. 示例代码

```
import unittest
from HTMLTestRunner import HTMLTestRunner
suite = unittest.TestLoader().discover("./", "test*.py")
f = open("test01.html", "wb")
runner = HTMLTestRunner(stream=f, title="自动化测试报告",
description="Chrome 浏览器")
runner.run(suite)
f.close()
```