unittest 框架介绍

- 框架

```
1. framework
2. 为了解决一类事情的功能集合
```

- Unittest框架

```
是Python自带的单元测试框架
-- 自带的,可以直接使用,不需要单外安装
-- 测试人员用来做自动化测试，作为自动化测试的执行框架,即管理和执行用例的
```

- 使用原因

```
1. 能够组织多个用例去执行
2. 提供丰富的断言方法
3. 能够生成测试报告
```

- 核心要素（组成）

```
1. TestCase：测试用例，这个测试用例是unittest 的组成部分,作用是用来书写真正的用例代码(脚本)
2. Testsuite：测试套件，作用是用来组装(打包)TestCase(测试用例)的,即可以将多个用例脚本文件组装到一起
3. TestRunner：测试执行(测试运行)，作用是用例执行TestSuite(测试套件)的
4. TestLoader：测试加载，是对TestSuite(测试套件) 功能的补充，作用是用来组装(打包)Testcase(测试用例)的
5. Fixture：测试夹具，是一种代码结构，书写前置方法(执行用例之前的方法)代码和后置方法(执行用例之后的方法)代码，即用例执行顺序前置--->用例--->后置
```

TestCase 测试用例

> 书写真正的用例脚本
>
> 单独一个测试用例，也是可以执行的

- 步骤

```
1. 导包unittest
2. 定义测试类,需要继承unittest.Testcase类,习惯性类名以Test开头
3. 书写测试方法，必须以test开头
4. 执行
```

```
"""
学习 TestCase(测试用例) 的使用
"""
# 1. 导包 unittest
No. 6 / 23
import unittest
# 2. 定义测试类，只要继承 unittest.TestCase 类，就是
测试类
class TestDemo(unittest.TestCase):
# 3. 书写测试方法，方法中的代码就是真正用例代码，
方法名必须以 test 开头
def test_method1(self):
print('测试方法一')
def test_method2(self):
print('测试方法二')
# 4. 执行
# 4.1 在类名或者方法名后边右键运行
# 4.1.1 在类名后边，执行类中的所有的测试方法
# 4.1.2 在方法名后边，只执行当前的测试方法
# 4.1 在主程序使用使用 unittest.main() 来执行，
if __name__ == '__main__':
unittest.main(
```

TestSuite和TestRunner

TestSuite（测试套件）

> 将多条用例脚本集合在一起,就是套件，即用来组装用例的

1. 导包unittest
2. 实例化套件对象unittest.Testsuite()
3. 添加用例方法

TestRunner（测试执行）

> 用来执行套件对象

1. 导包unittest
2. 实例化执行对象unittest.TextTestRunner()
3. 执行对象执行套件对象执行对象.run(套件对象)

整体步骤

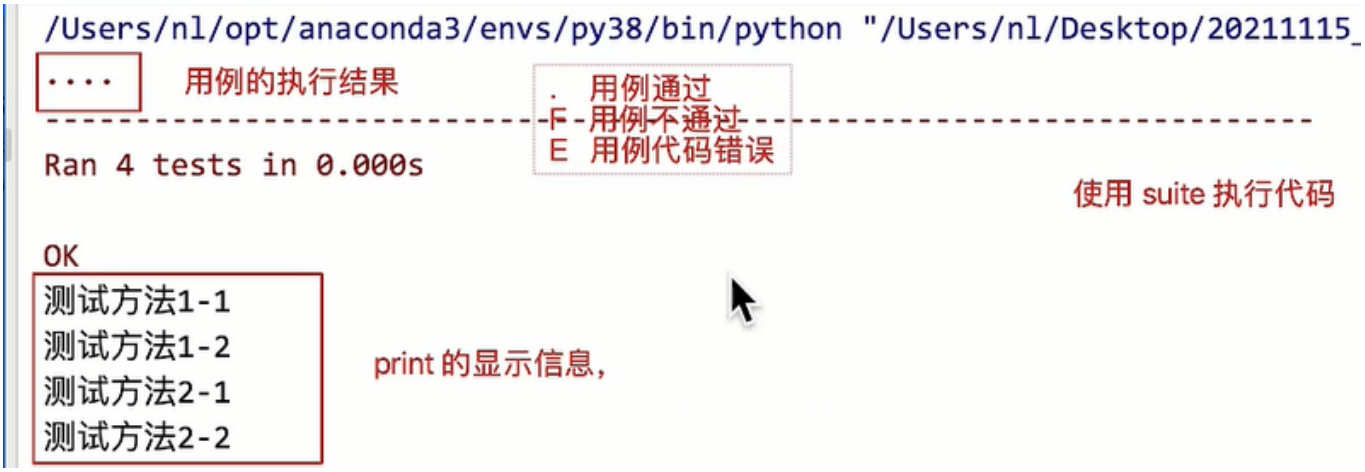
1. 导包unittest
2. 实例化套件对象unittest.Testsuite()
3. 添加用例方法
4. 实例化执行对象unittest.TextTestRunner()
5. 执行对象执行套件对象执行对象. run(套件对象)

```
# 1. 导包 unittest
import unittest
# 2. 定义测试类，只要继承 unittest.TestCase 类，就是
测试类
class TestDemo1(unittest.TestCase):
# 3. 书写测试方法，方法中的代码就是真正用例代码，
方法名必须以 test 开头
def test_method1(self):
print('测试方法1-1')
def test_method2(self):
print('测试方法1-2')
```

套件和执行

```
# 1. 导包 unittest
import unittest
from hm_02_testcase1 import TestDemo1
from hm_02_testcase2 import TestDemo2
# 2. 实例化套件对象 unittest.TestSuite()
suite = unittest.TestSuite()
# 3. 添加用例方法
# 3.1 套件对象.addTest(测试类名('测试方法名')) # 建议复制
suite.addTest(TestDemo1('test_method1'))
suite.addTest(TestDemo1('test_method2'))
suite.addTest(TestDemo2('test_method1'))
suite.addTest(TestDemo2('test_method2'))
# 4. 实例化 执行对象 unittest.TextTestRunner()
runner = unittest.TextTestRunner()
# 5. 执行对象执行 套件对象 执行对象.run(套件对象)
runner.run(suite)
# 套件对象.addTest(unittest.makeSuite(测试类名)) # 在
不同的 Python 版本中,可能没有提示
suite.addTest(unittest.makeSuite(TestDemo1))
suite.addTest(unittest.makeSuite(TestDemo2))
```

查看执行结果



TestLoader 测试加载

> 作用和TestSuite 作用一样,组装用例代码，同样也需要使用sTextTestRunner()去执行
>
> 10个用例脚本makeSuite()

1. 导包unittest
2. 实例化加载对象并加载用例--->得到的是套件对象
3. 实例化执行对象并执行

```
# 实例化加载对象并加载用例--->得到的是套件对象
import unittest
# suite = unittest.TestLoader().discover('用例所在的目录', '用例代码文件名*.py')
suite = unittest.TestLoader().discover('.', 'gz_02_testcase*.py')

# 实例化执行对象并执行
# runner = unittest.TextTestRunner()
# runner.run(suite)

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

练习

练习1

1. 创建一个目录case,作用就是用来存放用例脚本,
2. 在这个目录中创建5个用例代码文件, test_case1.py ...
3. 使用TestLoader去执行用例

将来的代码用例都是单独的目录中存放的

test_项目_模块_功能.py

练习2

1. 定义一个 tools模块，在这个模块中定义 add 的方法,可以对两个数字求和,返回求和结果
2. 书写用例，对add()函数进行测试

1, 1, 2
1, 2, 3
3, 4, 7
4, 5, 9

之前的测试方法,直接一个print
这个案例中的测试方法,调用add 函数，使用if 判断,来判断预期结果和实际结果是否相符预期结果2 3 7 9
实际结果调用add()

Fixture

> **代码结构,在用例执行前后会自动执行的代码结构**
>
> tpshop登录
> 1. 打开浏览器（一次)
> 2. 打开网页,点击登录（每次)
> 3. 输入用户名密码验证码1,点击登录(每次，测试方法)
> 4. 关闭页面(每次)
> 2. 打开网页,点击登录(每次)
> 3. 输入用户名密码验证码2,点击登录(每次，测试方法)

4. 关闭页面(每次)
2. 打开网页,点击登录（每次)
3. 输入用户名密码验证码3,点击登录（每次，测试方法)
4. 关闭页面(每次)
5. 关闭浏览器（一次)

方法级别Fixture

> 在每个用例执行前后都会自动调用,方法名是固定的

```
def setUp(self):        # 前置
# 每个用例执行之前都会自动调用pass
    pass
def tearDown(self):     # 后置
#每个用例执行之后都会自动调用pass
    pass

# 方法前置用例方法后置
# 方法前置用例方法后置
```

类级别Fixture

> 在类中所有的测试方法执行前后会自动执行的代码，只执行一次

```
# 类级别的 Fixture需要写作类方法@classmethod
def setUpclass(cls):    # 类前置
    pass

@classmethod
def tearDownclass(cls)  :   # 后置
    pass

# 类前置 方法前置 用例 方法后置 方法前置 用例 方法后置 类后置
```

模块级别 Fixture(了解)

> 模块级别在这个代码文件执行前后执行一次

```
# 在类外部定义函数
def setUpModule():
pass
def tearDownModule():
pass
import unittest
class TestLogin(unittest.TestCase):
def setUp(self) -> None:
print('2. 打开网页, 点击登录')
def tearDown(self) -> None:
print('4. 关闭网页')
@classmethod
def setUpClass(cls) -> None:
print('1. 打开浏览器')
@classmethod
def tearDownClass(cls) -> None:
print('5. 关闭浏览器')
def test_1(self):
print('3. 输入用户名密码验证码1,点击登录 ')
def test_2(self):
print('3. 输入用户名密码验证码2,点击登录 ')
def test_3(self):
print('3. 输入用户名密码验证码3,点击登录 ')
```

断言

> 前提:
>
> 用例脚本中
>
> 断言(使用代码自动的判断预期结果和实际结果是否相符)
>
> 参数化(将测试数据定义到json文件，使用)
>
> 跳过(某些用例由于某种原因不想执行，设置为跳过)
>
> 生成测试报告(suite和runner(第三方))

使用代码自动的判断预期结果和实际结果是否相符

使用代码自动的判断预期结果和实际结果是否相符assertEqual(预期结果,实际结果)
-- 判断预期结果和实际结果是否相等，如果相等，用例通过，如果不相等，抛出异常，用例不通过

assertIn(预期结果,实际结果)
-- 判断预期结果是否包含在实际结果中，如果存在，用例通过，如果不存在，抛出异常，用例不通过

```
import unittest
class TestAssert(unittest.TestCase):
def test_equal_1(self):
self.assertEqual(10, 10) # 用例通过
def test_assert_2(self):
self.assertEqual(10, 11) # 用例不通过
def test_in(self):
# self.assertIn('admin', '欢迎 admin 登录') # 包含 通过
# self.assertIn('admin', '欢迎 adminnnnnnnn 登录') # 包含 通过
# self.assertIn('admin', '欢迎 aaaaaadminnnnnnnnn 登录') # 包含 通过
# self.assertIn('admin', '欢迎 adddddmin 登录') # 不包含 不通过
self.assertIn('admin', 'admin') # 包含 通过
```

```
import unittest
from hm_02_assert import TestAssert
suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(TestAssert))
unittest.TextTestRunner().run(suite)
```

参数化

- 通过参数的方式来传递数据，从而实现数据和脚本分离。并且可以实现用例的重复执行。(在书写用例方法的时候，测
试数据使用变量代替，在执行的时候进行据说传递)
- 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

环境准备

```
因为参数化的插件 不是 unittest 自带的，所以想要使用 需要进行安装
Python 中 包(插件，模块) 的安装，使用 pip 工具
pip install parameterized
pip install -i https://pypi.douban.com/simple/ parameterized
# 在终端(cmd)中执行
```

```
(venv) PS D:\Code\Python\OOP> pip install parameterized
Collecting parameterized
  Downloading parameterized-0.9.0-py2.py3-none-any.whl (20 kB)
Installing collected packages: parameterized
Successfully installed parameterized-0.9.0
```

使用

```
1. 导包 from para... import para...
2. 修改测试方法，将测试方法中的测试数据使用 变量表示
3. 组织测试数据，格式 [(), (), ()]，一个元组就是一组测试数据
4. 参数化，在测试方法上方使用装饰器 @parameterized.expand(测试数据)
5. 运行(直接 TestCase 或者 使用 suite 运行)
```

```
import unittest
from tools import add
练习
json 文件
读取 json 文件
代码文件
from parameterized import parameterized
data = [(1, 1, 2), (1, 2, 3), (2, 3, 5), (4, 5, 9)]
class TestAdd(unittest.TestCase):
@parameterized.expand(data)
def test_add(self, a, b, expect):
print(f'a:{a}, b:{b}, expect: {expect}')
self.assertEqual(expect, add(a, b))
if __name__ == '__main__':
unittest.main()
```

测试报告

> 使用第三方的报告模版，生成报告 HTMLTestReport, 本质是 TestRunner
>
> - 安装
>   pip install -i https://pypi.douban.com/simple/ HTMLTestReport
> - 使用
>
> 1. 导包 unittest、HTMLTestReport
> 2. 组装用例(套件， loader)
> 3. 使用 HTMLTestReport 中的 runner 执行套件
> 4. 查看报告

```
import unittest
from htmltestreport import HTMLTestReport
from hm_04_pa1 import TestAdd
# 套件
suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(TestAdd))
# 运行对象
# runner = HTMLTestReport(报告的文件路径后缀.html, 报告的标题, 其他的描述信息)
runner = HTMLTestReport('test_add_report.html', '加法用例测试报告', 'xxx')
runner.run(suite)
```