

Table of Contents

软件测试Python课程	1.1
UnitTest框架	1.2
UnitTest基本使用	1.2.1
Fixture	1.2.2
断言	1.2.3
参数化	1.2.4
跳过	1.2.5
生成HTML测试报告	1.2.6
附件-断言资料	1.2.7

软件测试Python课程

本阶段课程不仅可以帮助我们进入Python语言世界，同时也是后续UI自动化测试、接口自动化测试等课程阶段的语言基础。

Life is short, you need Python! -- 人生苦短，我用Python！

课程大纲

序号	章节	知识点
1	Python基础	1. 认识Python 2. Python环境搭建 3. PyCharm 4. 注释、变量、变量类型、输入输出、运算符
2	流程控制结构	1. 判断语句 2. 循环
3	数据序列	1. 字符串 2. 列表 3. 元组 4. 字典
4	函数	1. 函数基础 2. 变量进阶 3. 函数进阶 4. 匿名函数
5	面向对象	1. 面向对象编程介绍 2. 类和对象 3. 面向对象基础语法 4. 封装、继承、多态 5. 类属性和类方法
6	异常、模块、文件操作	1. 异常 2. 模块和包 3. 文件操作
7	UnitTest框架	1. UnitTest基本使用 2. UnitTest断言 3. 参数化 4. 生成HTML测试报告

课程目标

1. 掌握如何搭建Python开发环境；
2. 掌握Python基础语法, 具备基础的编程能力；
3. 建立编程思维以及面向对象程序设计思想；
4. 掌握如何通过UnitTest编写测试脚本，并生成HTML测试报告。

UnitTest框架

目标

1. 掌握UnitTest框架的基本使用方法
2. 掌握断言的使用方法
3. 掌握如何实现参数化
4. 掌握测试报告的生成

UnitTest基本使用

目标

1. 掌握UnitTest框架的基本使用方法

1. UnitTest框架

1.1 什么是框架？

说明：

1. 框架英文单词framework
2. 为解决一类事情的功能集合

1.2 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。

1.3 为什么使用UnitTest框架？

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

1.4 UnitTest核心要素

1. TestCase
2. TestSuite
3. TestRunner
4. TestLoader
5. Fixture

2. TestCase

说明：`TestCase`就是测试用例

2.1 案例

定义一个实现加法操作的函数，并对该函数进行测试

2.2 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

2.3 执行测试用例

方式一：

使用`pycharm`在代码上点击鼠标右键，选择使用`UnitTest`运行

方式二：

调用 `unittest.main()` 来运行

思考：如何同时运行多个测试用例？

3. 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(TestCase("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`TestRunner`才能被执行

4. 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

需求

将`test01.py..test10.py`共10条用例, 将这10条用例批量执行;

问题

1. 使用`suite.addtest(unittest.makeSuite(className))`导入10条测试类
2. `.addtest()`需要添加10次

5. 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, 默认为`'test*.py'`

也可以使用`unittest.defaultTestLoader` 代替 `unittest.TestLoader()`

运行:

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

5.1 TestLoader与TestSuite区别

1. `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)
2. `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;

6. 总结

1. `UnitTest`框架的作用？
2. 如何定义测试用例？
3. 如何执行测试用例？
4. 如何使用`TestSuite`？
5. 如何运行`TestSuite`？
6. 如何使用`TestLoader`？
7. `TestLoader`与`TestSuite`的区别？

Fixture

目标

1. 掌握方法级别和类级别的Fixture
2. 了解模块级别的Fixture

1. Fixture

小需求：在一个测试类中定义多个测试方法，查看每个测试方法执行完所花费的时长。

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

Fixture控制级别：

1. 方法级别
2. 类级别
3. 模块级别

1.1 方法级别

使用：

1. 初始化(前置处理):
`def setUp(self)` --> 首先自动执行
2. 销毁(后置处理):
`def tearDown(self)` --> 最后自动执行
3. 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

1.2 类级别

使用：

1. 初始化(前置处理):
`@classmethod`
`def setUpClass(cls):` --> 首先自动执行
2. 销毁(后置处理):
`@classmethod`
`def tearDownClass(cls):` --> 最后自动执行
3. 运行于测试类的始末，即：每个测试类只会运行一次setUpClass和tearDownClass

1.3 模块级别 [了解]

使用:

1. 初始化(前置处理):
`def setUpModule()` --> 首先自动执行
2. 销毁(后置处理):
`def tearDownModule()` --> 最后自动执行
3. 运行于整个模块的始末, 即: 整个模块只会运行一次`setUpModule`和`tearDownModule`

断言

目标

1. 理解什么是断言
2. 掌握断言`assertEqual`、`assertIn`方法
3. 了解`unittest`其他断言方法

1. unittest断言

1.1 什么是断言？

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

1.2 为什么要学习断言？

1. 提高测试效率
2. 实现自动化测试（让脚步在无人值守状态下运行）

1.3 unittest断言方法

说明：

1. `unittest`中提供了非常丰富的断言方法，请参考附件资料
2. 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

2. 常用的unittest断言方法

序号	断言方法	断言描述
1	<code>assertTrue(expr, msg=None)</code>	验证 <code>expr</code> 是 <code>true</code> ，如果为 <code>false</code> ，则fail
2	<code>assertFalse(expr, msg=None)</code>	验证 <code>expr</code> 是 <code>false</code> ，如果为 <code>true</code> ，则fail
3	<code>assertEqual(expected, actual, msg=None)</code>	验证 <code>expected==actual</code> ，不等则fail 【掌握】
4	<code>assertNotEqual(first, second,</code>	验证 <code>first != second</code> ，相等则fail

4	<code>msg=None)</code>	验证 <code>first != second</code> , 相等则fail
5	<code>assertIsNone(obj, msg=None)</code>	验证obj是None, 不是则fail
6	<code>assertIsNotNone(obj, msg=None)</code>	验证obj不是None, 是则fail
7	<code>assertIn(member, container, msg=None)</code>	验证是否member in container 【掌握】
8	<code>assertNotIn(member, container, msg=None)</code>	验证是否member not in container

2.1 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。

```
import unittest

def add(x, y):
    return x + y

class TestAssert(unittest.TestCase):

    def test01(self):
        num = add(1, 2)
        self.assertEqual(3, num)

    def test02(self):
        num = add(1, 2)
        is_ok = num == 3
        self.assertTrue(is_ok)
```

参数化

目标

1. 掌握如何实现参数化

1. 参数化

1.1 小需求

需求：定义一个实现加法操作的函数，并对该函数进行测试

示例代码

```
import unittest

# 求和
def add(x, y):
    return x + y

class TestAdd(unittest.TestCase):
    def test_add_01(self):
        result = add(1, 1)
        self.assertEqual(result, 2)

    def test_add_02(self):
        result = add(1, 0)
        self.assertEqual(result, 1)

    def test_add_03(self):
        result = add(0, 0)
        self.assertEqual(result, 0)

    def test_add(self):
        test_data = [(1, 1, 2), (1, 0, 1), (0, 0, 0)]
        for x, y, expect in test_data:
            print("x={} y={} expect={}".format(x, y, expect))
            result = add(x, y)
            self.assertEqual(result, expect)
```

发现问题

1. 一条测试数据定义一个测试函数，代码冗余度太高
2. 一个测试函数中测试多条数据，最终只会有一个测试结果

1.2 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。
`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
@parameterized.expand([(1, 1, 2), (1, 0, 1), (0, 0, 0)])
def test_add(self, x, y, expect):
    pass

# 方式二
data = [(1, 1, 2), (1, 0, 1), (0, 0, 0)]
@parameterized.expand(data)
def test_add(self, x, y, expect):
    pass

# 方式三
def build_data():
    return [(1, 1, 2), (1, 0, 1), (0, 0, 0)]

@parameterized.expand(build_data)
def test_add(self, x, y, expect):
    pass
```

示例代码

```
import unittest
from parameterized import parameterized

# 求和
```

```

def add(x, y):
    return x + y

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

class TestAdd(unittest.TestCase):
    @parameterized.expand([(1, 1, 2), (1, 0, 1), (0, 0, 0)])
    def test_add_1(self, x, y, expect):
        print("x={} y={} expect={}".format(x, y, expect))
        result = add(x, y)
        self.assertEqual(result, expect)

    data = [(1, 1, 2), (1, 0, 1), (0, 0, 0)]

    @parameterized.expand(data)
    def test_add_2(self, x, y, expect):
        print("x={} y={} expect={}".format(x, y, expect))
        result = add(x, y)
        self.assertEqual(result, expect)

    @parameterized.expand(build_data)
    def test_add_3(self, x, y, expect):
        print("x={} y={} expect={}".format(x, y, expect))
        result = add(x, y)
        self.assertEqual(result, expect)

```

跳过

目标

1. 掌握如何把测试函数标记成跳过

1. 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。

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)
```

示例代码

```
import unittest

version = 35

class TestSkip(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 TestSkip2(unittest.TestCase):

    def test_a(self):
        print("test_a")

    def test_b(self):
```

```
print("test_b")
```


生成HTML测试报告

目标

1. 掌握如何生成HTML测试报告方法

1. 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

1.1 为什么要生成测试报告

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

1.2 unittest生成HTML报告的方式

1. Export Test Results （unittest 自带）
2. HTMLTestRunner（第三方模板）【重点】

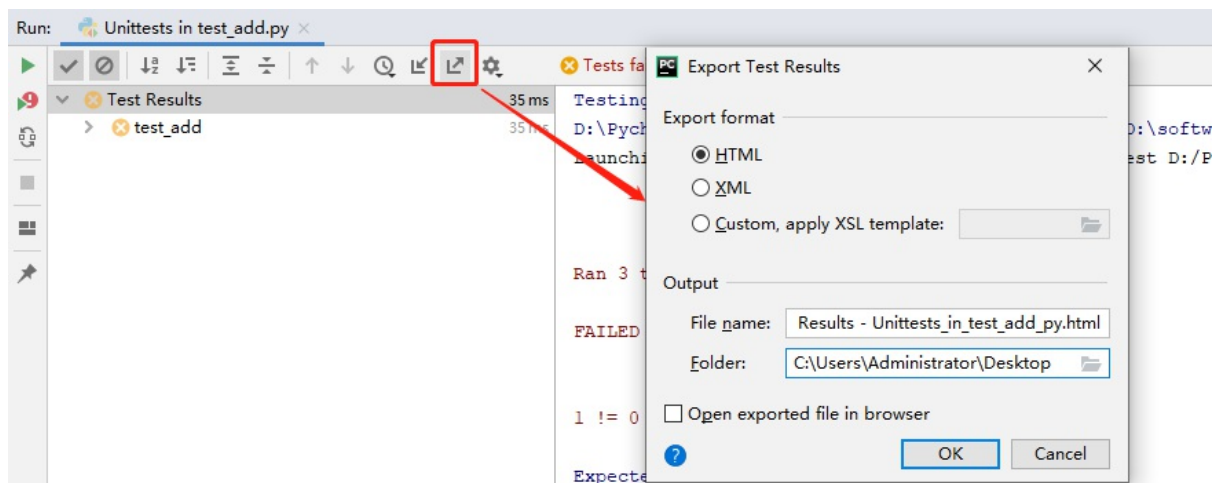
2. Export Test Results （自带）

2.1 测试报告截图

Unittests in test_add.py: 3 total, 3 passed			25 ms
			Collapse Expand
test_add			25 ms
TestAdd			25 ms
test_add_01	passed		25 ms
test_add_02	passed		0 ms
test_add_03	passed		0 ms

Generated by PyCharm on 2050/4/15 下午2:48

2.2 自带报告生成操作图



3. HTMLTestRunner【重点】

3.1 HTMLTestRunner介绍

- HTMLTestRunner是一个第三方的类库，用来执行测试用例并生成HTML格式的测试报告
- 网上有多种不同的实现版本，用法都类似，只是生成的HTML报告样式有差别
- 注意：下载的文件要与Python的版本兼容（有些版本只支持Python2.x）

本课程中介绍两种模板文件，HTMLTestRunner.py 和 HTMLTestRunner_PY3.py

HTMLTestRunner.py：样式比较简单，不需要依赖外部JS和CSS文件

HTMLTestRunner_PY3.py：样式比较美观，需要依赖外部JS和CSS文件

3.2 测试报告截图

HTMLTestRunner

单元测试报告

Start Time: 2050-04-15 15:08:30
Duration: 0:00:00.000997
Status: Pass 2 Failure 1

V1.0

Show [Summary](#) [Failed](#) [All](#)

Test Group/Test case	Count	Pass	Fail	Error	View
unit.test_add.TestAdd	3	2	1	0	Detail
test_add_01			pass		
test_add_02			pass		
test_add_03			fail		
<div>ftl.3: Traceback (most recent call last): File "D:\ProharmProjects\python\unit\test_add.py", line 20, in test_add_03 self.assertEqual(result, 1) AssertionError: 0 != 1</div>					
Total	3	2	1	0	

HTMLTestRunner_PY3

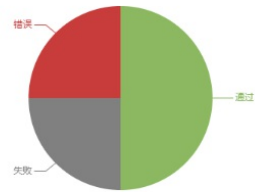
单元测试报告

开始时间: 2020-04-16 16:11:02
运行时长: 0:00:00.012177
测试结果: 通过 2 失败 1 错误 1 通过率= 50.00%

V1.0

通过
失败
错误

测试执行情况



概要(50.00%) 未通过(2) 通过(2) 所有(4)

用例集/测试用例	总数	通过	失败	错误	查看
unit.test_add.TestAdd	4	2	1	1	详细
test_add_03			失败		
test_error				错误	
总计	4	2	1	1	通过率: 50.00%

3.3 测试报告 生成步骤分析

1. 复制 HTMLTestRunner.py 或 HTMLTestRunner_PY3.py 文件到项目文件夹
2. 导入HTMLTestRunner、unittest包
3. 生成测试套件

```
suite = unittest.TestSuite()  
suite.addTest(TestLogin("test_login"))  
# 或者  
suite = unittest.defaultTestLoader.discover(test_dir, pattern="test*.py")
```

4. 设置报告生成路径和文件名

```
file_name = "./report/report.html"
```

5. 打开报告

```
with open(file_name, 'wb') as f:  
    pass
```

6. 实例化HTMLTestRunner对象:

```
runner = HTMLTestRunner(stream=f,[verbosity],[title],[description])
```

参数说明:

stream: 文件流, 打开写入报告的名称及写入编码格式)

verbosity: [可选参数], 默认值为1, 不显示完整结果(用例成功输出'.', 失败输出'F', 错误输出'E')

值为2时, 需要打印详细的返回信息

title: [可选参数], 为报告标题, 如XXX测试报告

description: [可选参数], 为报告描述信息

7. 执行: runner.run(suite)

3.4 实现代码

```
import unittest  
  
# from unit.lib.HTMLTestRunner import HTMLTestRunner  
from unit.lib.HTMLTestRunner_PY3 import HTMLTestRunner  
from unit.test_add import TestAdd  
  
# 构建测试套件  
suite = unittest.TestSuite()  
suite.addTest(unittest.makeSuite(TestAdd))  
  
# 创建HTMLTestRunner  
report_path = "./report.html"  
with open(report_path, "wb") as f:  
    runner = HTMLTestRunner(f, verbosity=2, title="单元测试报告", description="V1.0")  
    runner.run(suite)
```

附件-断言资料

UnitTest断言方法

序号	断言方法	断言描述
1	<code>assertEqual(arg1, arg2, msg=None)</code>	验证 <code>arg1=arg2</code> ，不等则fail 【常用】
2	<code>assertNotEqual(arg1, arg2, msg=None)</code>	验证 <code>arg1 != arg2</code> ，相等则fail
3	<code>assertTrue(expr, msg=None)</code>	验证 <code>expr</code> 是 <code>true</code> ，如果为 <code>false</code> ，则fail 【常用】
4	<code>assertFalse(expr, msg=None)</code>	验证 <code>expr</code> 是 <code>false</code> ，如果为 <code>true</code> ，则fail 【常用】
5	<code>assertIs(arg1, arg2, msg=None)</code>	验证 <code>arg1</code> 、 <code>arg2</code> 是同一个对象，不是则fail
6	<code>assertIsNot(arg1, arg2, msg=None)</code>	验证 <code>arg1</code> 、 <code>arg2</code> 不是同一个对象，是则fail
7	<code>assertIsNone(expr, msg=None)</code>	验证 <code>expr</code> 是 <code>None</code> ，不是则fail
8	<code>assertIsNotNone(expr, msg=None)</code>	验证 <code>expr</code> 不是 <code>None</code> ，是则fail
9	<code>assertIn(arg1, arg2, msg=None)</code>	验证 <code>arg1</code> 是 <code>arg2</code> 的子串，不是则fail
10	<code>assertNotIn(arg1, arg2, msg=None)</code>	验证 <code>arg1</code> 不是 <code>arg2</code> 的子串，是则fail
11	<code>assertIsInstance(obj, cls, msg=None)</code>	验证 <code>obj</code> 是 <code>cls</code> 的实例，不是则fail
12	<code>assertNotIsInstance(obj, cls, msg=None)</code>	验证 <code>obj</code> 不是 <code>cls</code> 的实例，是则fail
13	<code>assertAlmostEqual (first, second, places = 7, msg = None, delta = None)</code>	验证 <code>first</code> 约等于 <code>second</code> 。 <code>places</code> : 指定精确到小数点后多少位，默认为7
14	<code>assertNotAlmostEqual (first, second, places, msg, delta)</code>	验证 <code>first</code> 不约等于 <code>second</code> 。 <code>places</code> : 指定精确到小数点后多少位，默认为7 注：在上述的两个函数中，如果 <code>delta</code> 指定了值，则 <code>first</code> 和 <code>second</code> 之间的差值必须 $\leq \delta$
15	<code>assertGreater (first, second, msg = None)</code>	验证 <code>first > second</code> ，否则fail
16	<code>assertGreaterEqual (first, second, msg = None)</code>	验证 <code>first ≥ second</code> ，否则fail
17	<code>assertLess (first, second, msg = None)</code>	验证 <code>first < second</code> ，否则fail

18	<code>assertLessEqual (first, second, msg = None)</code>	验证 <code>first ≤ second</code> ，否则 <code>fail</code>
19	<code>assertRegexpMatches (text, regexp, msg = None)</code>	验证正则表达式 <code>regexp</code> 搜索匹配的文本 <code>text</code> 。 <code>regexp</code> : 通常使用 <code>re.search()</code>
20	<code>assertNotRegexpMatches (text, regexp, msg = None)</code>	验证正则表达式 <code>regexp</code> 搜索不匹配的文本 <code>text</code> 。 <code>regexp</code> : 通常使用 <code>re.search()</code> 说明: 两个参数进行比较 (>、≥、<、≤、约等、不约等)
21	<code>assertListEqual(list1, list2, msg = None)</code>	验证列表 <code>list1</code> 、 <code>list2</code> 相等，不等则 <code>fail</code> ，同时报错信息返回具体的不同的地方
22	<code>assertTupleEqual (tuple1, tuple2, msg = None)</code>	验证元组 <code>tuple1</code> 、 <code>tuple2</code> 相等，不等则 <code>fail</code> ，同时报错信息返回具体的不同的地方
23	<code>assertSetEqual (set1, set2, msg = None)</code>	验证集合 <code>set1</code> 、 <code>set2</code> 相等，不等则 <code>fail</code> ，同时报错信息返回具体的不同的地方
24	<code>assertDictEqual (expected, actual, msg = None)</code>	验证字典 <code>expected</code> 、 <code>actual</code> 相等，不等则 <code>fail</code> ，同时报错信息返回具体的不同的地方