day09 课堂笔记

课程之前

复习和反馈

作业

今日内容

```
unittest 框架，自动化测试中使用，管理执行测试用例的
- unittest 框架的基本使用方法(组成)
- 断言的使用 (让程序自动的判断预期结果和实际结果是否相符)
- 参数化(多个测试数据，测试代码写一份 传参)
- 生成测试报告
```

复习

```
python 基础，搭建环境，注释，变量，标识符(由字母,数字和 下划线组成,不能以数字开头)
关键字: True False None and or not in if elif else break continue while for def return lambda global pass class from  import as try except finall is raise
```

```
input() ---> str 类型
类型转换: int() float() str()  list() tuple()
print()
运算符 %  //
判断 和 循环
if 判断条件:
    条件成立执行的代码
else:
    条件不成立执行的代码

循环:
while True:
    pass

for 变量 in 容器:
    pass

for 变量 in range(循环的次数):
    pass

容器:
-   字符串 str 下标 切片
-   列表 list []  追加 列表.append()
-   元组 tuple (), (数据, )
-   字典 dict  {}, 字典.get('键')

函数:
```

```
def 函数名(参数,  *args, xx=oo):
    pass
    return
# 缺省参数
# 不定长参数(多值, 可变)  *args

lambda 参数: 表达式

面向对象:
class 类名:
    def __init__(self):
        # 添加属性

对象 = 类名()
对象.方法名()

继承:
class 子类(父类):
    pass

# 调用父类的方法, super().方法名()

文件操作
1. json 文件的定义
2. json 文件的读取
import json
```

```
with open('文件名', encoding='utf-8') as f:
    data = json.load(f)  # 列表 或 字典


异常:
try:
    可能发生异常的代码
except Exception as e:
    发生异常执行的代码
else:
    没有发生异常执行的代码
finally:
    不管有没有发生异常,都会执行的代码
```

unittest 框架的介绍

- 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

- 使用的原因

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

书写真正的用例代码(脚本)
单独一个测试用例 也是可以执行

- 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
```

- 注意事项

```
1. 代码文件名字 要满足标识符的规则
2. 代码文件名 不要使用中文
```

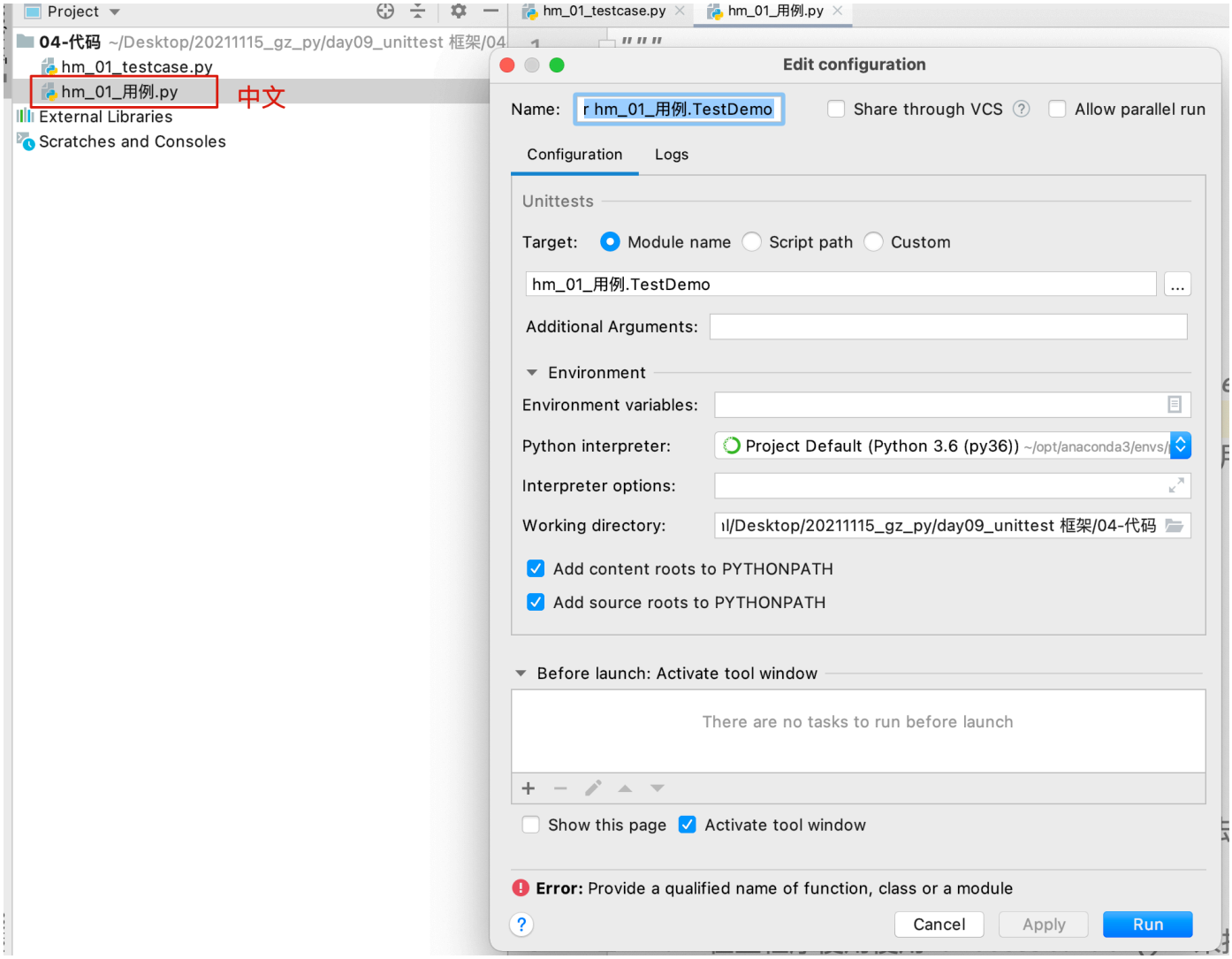
- 代码

```
"""
学习 TestCase(测试用例) 的使用
"""


# 1. 导包 unittest
```

```python
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()
```

可能出现的错误

文件名包含中文



右键运行 没有 `unittest for ....`

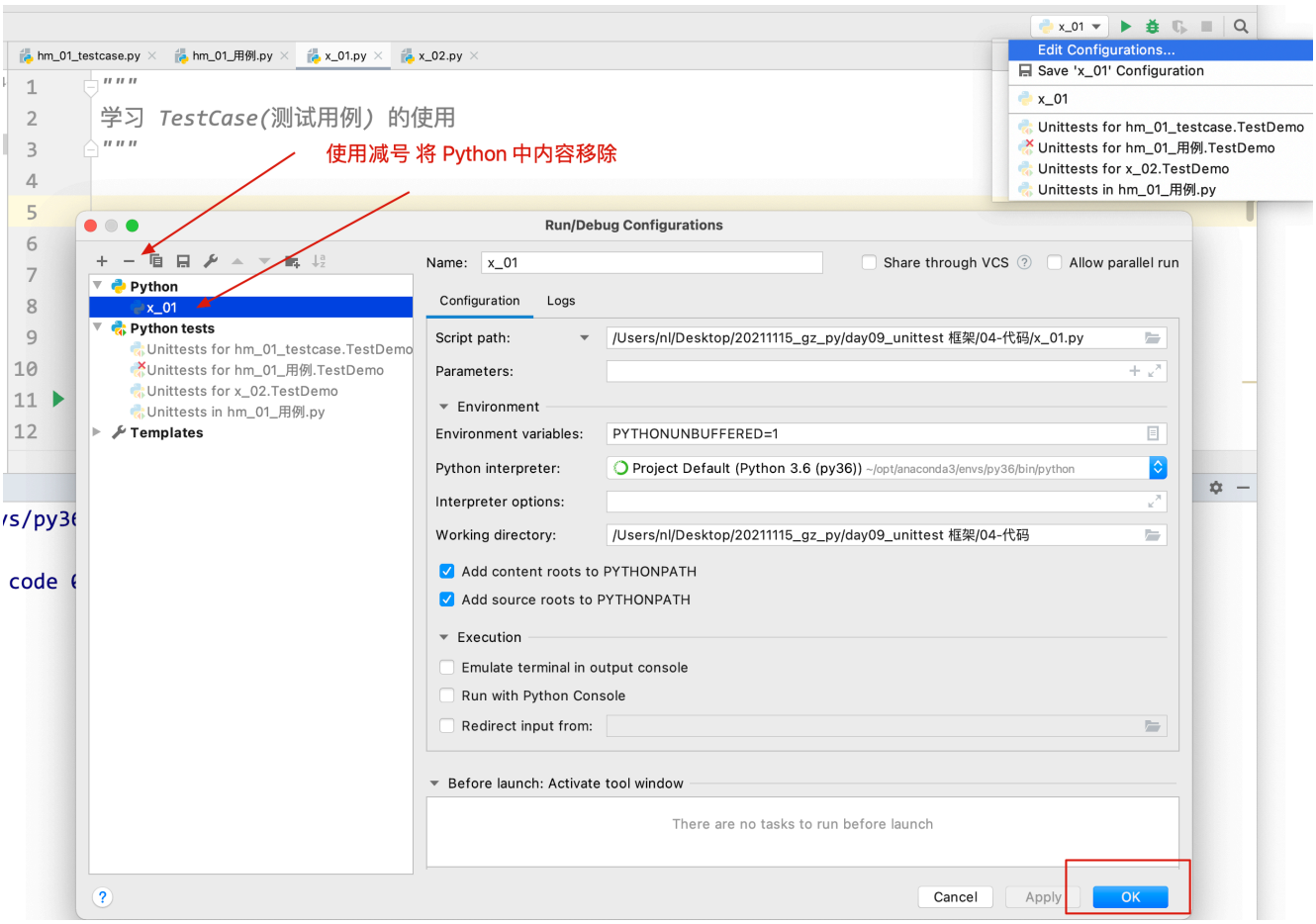
- 解决方案一

> 新建一个代码文件，将之前的代码复制过来

- 方法二

```
# 4.1 在主程序使用使用 unittest.main()  来执行,
if __name__ == '__main__':
    unittest.main()
```

- 方法三



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
3.1 套件对象.addTest(测试类名('测试方法名'))
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)
```

代码案例

```
套件可以用来组装用例，创建多个用例代码文件
```

- 用例代码文件

```python
# 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')
```

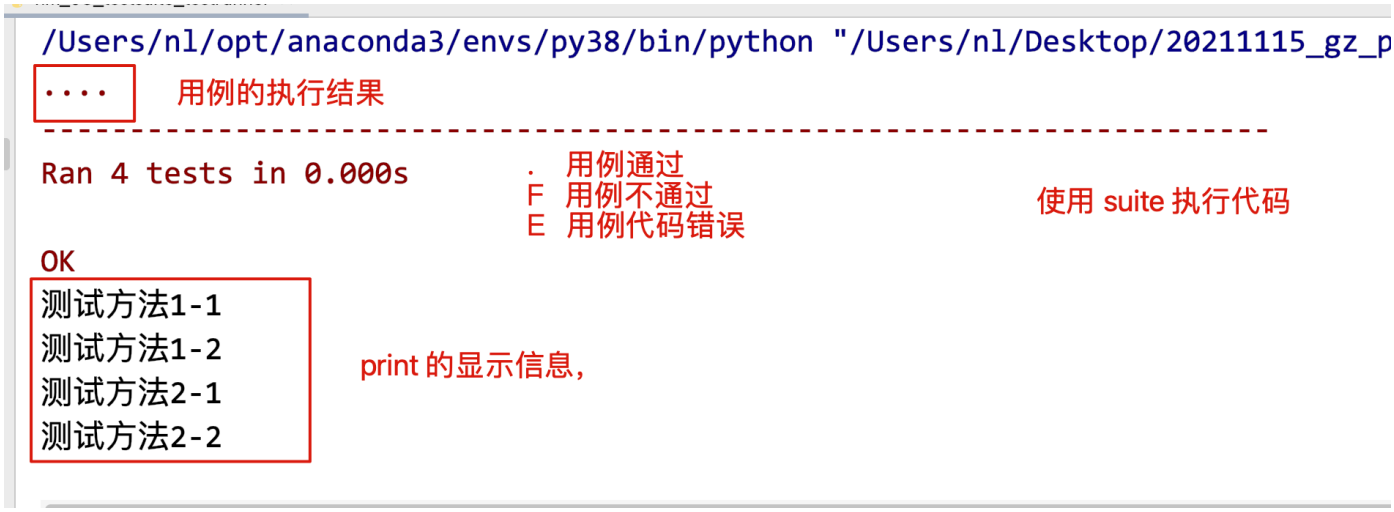
- 套件和执行

```python
# 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 TextTestRunner() 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

```python
import unittest

# 实例化加载对象并加载用例,得到套件对象
# suite = unittest.TestLoader().discover('用例所在的目录', '用例代码文件名*.py')
suite = unittest.TestLoader().discover('.', 'hm_02*.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()
```

```
def add(a, b):
    return a + b
```

- 用例代码

```
import unittest

from tools import add


class TestAdd(unittest.TestCase):
    def test_1(self):
        """1,1,2"""
        if 2 == add(1, 1):
            print(f'用例 {1}, {1}, {2}通过')
        else:
            print(f'用例 {1}, {1}, {2}不通过')

    def test_2(self):
        if 3 == add(1, 2):
            print(f'用例 {1}, {2}, {3}通过')
        else:
            print(f'用例 {1}, {2}, {3}不通过')

    def test_3(self):
```

```
        if 7 == add(3, 4):
            print(f'用例 {3}, {4}, {7}通过')
        else:
            print(f'用例 {3}, {4}, {7}不通过')

    def test_4(self):
        if 9 == add(4, 5):
            print(f'用例 {4}, {5}, {9}通过')
        else:
            print(f'用例 {4}, {5}, {9}不通过')
```

- suite 代码

```
import unittest

from hm_06_test_add import TestAdd

suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(TestAdd))

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

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

def tearDown(self):  # 后置
    # 每个用例执行之后 都会自动调用
    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，就是代码文件
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
```

```python
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 ')
```