day09 笔记

课程之前

复习和反馈

```
pytest ui 自动化课程中进行学习



在 Windows 中绝对路径路径可能存在的问题:
c:/xx/ddd
c:\xxx\xxx    可能会出现 \和后边的字符结合组成转义字符
c:\\xxx\xx
r"c:\xx\xx"
```

作业

```
先将 10 个数字, 转为字符串, 存入 列表
','.join(列表)
```

```
def func1():
    # 1. 获取用户输入的数字
    num = input('请输入数字:')
    # 2. 判断获取的数字是否整数
    try:
```

```
        num = int(num)
    # 3. 如果不是整数，提示输入错误
    except Exception as e:
        print("输入错误", e)
    # 4. 如果是整数，则进一步判断是奇数还是偶数
    else:
        if num % 2 == 0:
            print('偶数')
        else:
            print('奇数')
    # 5. 最终提示：程序运行结束
    finally:
        print('程序运行结束')


def func2():
    # 1. 获取用户输入的数字
    num = input('请输入数字:')
    # 2. 判断获取的数字是否整数
    if num.isdigit():
        # 类型转换
        num = int(num)
        # 4. 如果是整数，则进一步判断是奇数还是偶数
        if num % 2 == 0:
            print('偶数')
        else:
            print('奇数')
```

```
    # 3. 如果不是整数，提示输入错误
    else:
        print("输入错误")
    # 5. 最终提示：程序运行结束
    print('程序运行结束')
```

今日内容

- 异常
- 模块和包
  - 导入模块(导包)
  - `if __name__ == "__main__":`
- Unitest 框架的学习
  - 了解, 基本组成

异常

异常传递[了解]

异常传递是 Python 中已经实现好了,我们不需要操作, 我们知道异常会进行传递.

异常传递: 在函数嵌套调用的过程中, 被调用的函数 ,发生了异常,如果没有捕获,会将这个异常向外层传递. .... 如果传到最外层还没有捕获,才报错

模块和包

1. Python 源代码文件就是一个模块
2. 模块中定义的变量 函数 类,都可以让别人使用,同样,可以使用别人定义的(好处: 别人定义好的不需要我们再次书写,直接使用即可)
3. 想要使用 别人的模块中的内容工具(变量, 类, 函数), 必须先导入模块 才可以
4. 我们自己写的代码,想要作为模块使用,代码的名字需要满足标识符的规则(由数字,字母下划线组成,不能以数字开头)

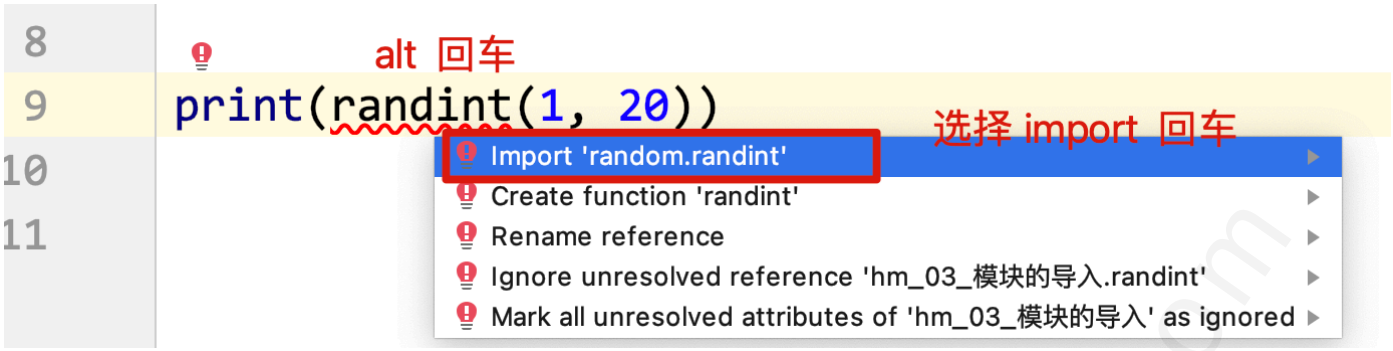
导入模块的语法

方式一

```
import 模块名
# 使用模块中的内容
模块名.工具名

# 举例
import random
import json
random.randint(a, b)
json.load()
json.dump()
```

方式二

```
from 模块名 import 工具名
# 使用
工具名  # 如果是函数和类需要加括号

# 举例
from random import randint
from json import load, dump
randint(a, b)
load()
dump()
```



方式三 [了解] 基本不用

```
from 模块名 import *  # 将模块中所有的内容都导入

from random import *
from json import *

randint(a, b)
load()
dump()
```

```
# 方式一 重点记忆
# import random
#
# print(random.randint(1, 20))

# 方式二, 重要快捷方式导包
# from random import randint
# from random import randint
#
```

```
# print(randint(1, 20))

# 方式三，问题：可能存在多个模块中有相同的名字的工具，会产生冲突
from random import *

print(randint(1, 20))
```

对于导入的模块和工具可以使用 as 关键字给其起别名
注意：如果起别名,原来的名字就不能用了，只能使用别名

模块的查找顺序

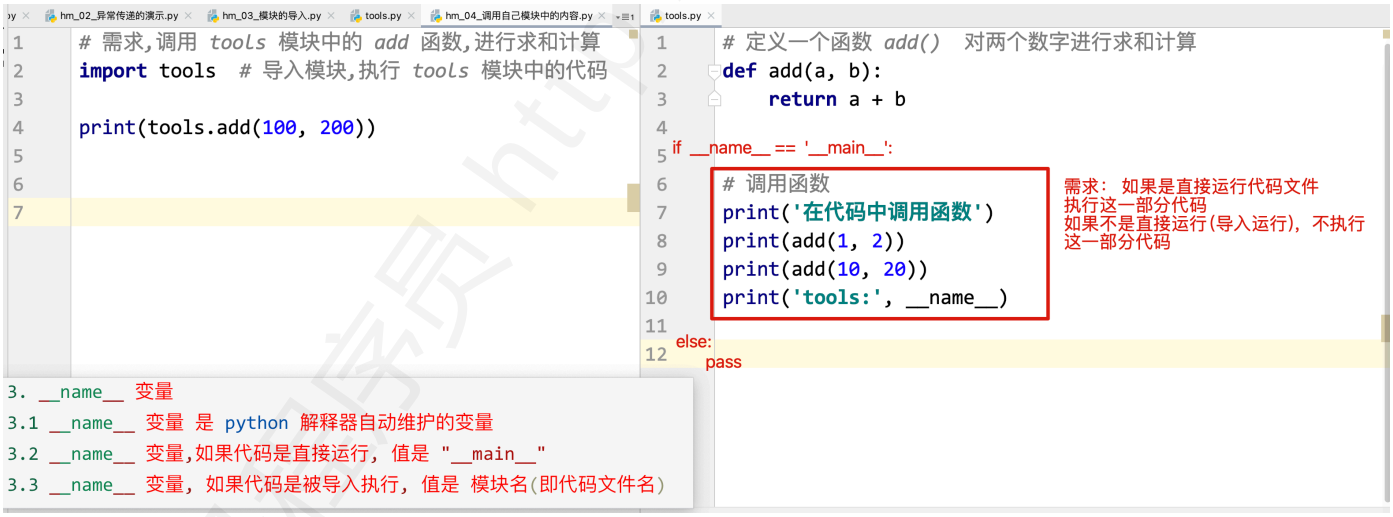
在导入模块的时候 会先在当前目录中找模块，如果找到,就直接使用
如果没有找到回去系统的目录中进行查找，找到,直接使用
没有找到，报错

注意点：
定义代码文件的时候，你的代码名字不能和你要导入的模块名字相同

__name__ 的作用

1. 每个代码文件都是一个模块
2. 在导入模块的时候，会执行模块中的代码(三种方法都会)
3. __name__ 变量
3.1 __name__ 变量 是 python 解释器自动维护的变量
3.2 __name__ 变量,如果代码是直接运行，值是 "__main__"
3.3 __name__ 变量，如果代码是被导入执行，值是 模块名(即代码文件名)

在代码文件中，在被导入时不想被执行的代码,可以写在 if __name__ == "__main__": 代码的缩进中



```python
# 需求,调用 tools 模块中的 add 函数,进行求和计算
import tools  # 导入模块,执行 tools 模块中的代码

print(tools.add(100, 200))
```

```python
# 定义一个函数 add()  对两个数字进行求和计算
def add(a, b):
    return a + b


if __name__ == '__main__':
    # 调用函数
    print('在代码中调用函数')
    print(add(1, 2))
    print(add(10, 20))
    print('tools:', __name__)
```

```
Run: hm_04_调用自己模块中的内容
/Users/nl/opt/anaconda3/envs/py36/bin/python /Users/nl/Desktop/20210620-23-python/day09_异常和模块/04-代
300
```

代码练习

1. 定义一个模块 tools.py
2. 在模块中定义一个函数，func，输出 '我是 tools 模块中的 funn 函数'
3. 在模块中定义一个类，Dog，具有属性 name，age，方法 play，输出 'xx 在快乐的玩耍'
4. 新建一个代码文件，调用 tools 模块中的 func 函数 并创建一个 Dog 类的对象，调用 play 方法

hm_05_练习.py

```python
import tools

# 调用函数
tools.func()

# 创建对象
dog = tools.Dog('小白', 2)
dog.play()
```

tools.py

```python
def func():
    print('我是 tools 模块中的 func 函数')


class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def play(self):
        print(f"{self.name} 在快乐的玩耍")
```

包(package)

在 Python 中，包 是一个目录，只不过在这个目录存在一个文件 __init__.py(可以是空的)
将功能相近或者相似的代码放在一起的.

在 Python 中使用的时候,不需要可以是区分是包还是模块，使用方式是一样的

random 模块 (单个代码文件)
json 包(目录)
unittest 包(目录)

1. import 包名
2. alt 回车 快捷导入

UnitTest框架

介绍

- 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

> 需要按照框架的规定(套路) 去书写代码

- 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。

自带的框架(官方): 不需要单外安装，只要安装了 Python，就可以使用
random, json, os, time
第三方框架: 想要使用 需要先安装后使用(pytest)
selenium , appium, requests

单元测试框架: 主要用来做单元测试，一般单元测试是开发做的.
对于测试来说，unittest 框架的作用是 自动化脚本(用例代码) 执行框架(使用 unittest 框架 来 管理 运行 多个测试用例的)

- 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(让程序代码代替人工自动的判断预期结果和实际结果是否相符)
3. 能够生成测试报告

- **UnitTest**核心要素(unitest 的组成部分)

1. TestCase(最核心的模块)

   TestCase(测试用例), 注意这个测试用例是 unittest 框架的组成部分, 不是手工和自动化中我们所说的用例(Test Case)

   主要作用: 每个 TestCase(测试用例) 都是一个代码文件, 在这个代码文件中 来书写 真正的用例代码

2. TestSuite

   TestSuite(测试套件), 用来 管理 组装(打包)多个 TestCase(测试用例) 的

3. TestRunner

   TestRunner(测试执行,测试运行), 用来 执行 TestSuite(测试套件)的

4. TestLoader

TestLoader(测试加载)，功能是对 TestSuite(测试套件) 功能的补充，
管理 组装(打包)多个 TestCase(测试用例) 的

5. Fixture

Fixture(测试夹具)，书写在 TestCase(测试用例) 代码中，是一个代码结构，可以在每个方法执行前后都会执行的内容

举例:
登录的测试用例，每个用例中重复的代码就可以写在 Fixture 代码结构中，只写一遍，但每次用例方法的执行,都会执行 Fixture 中的代码
1. 打开浏览器
2. 输入网址

TestCase(测试用例)

1. 是一个代码文件，在代码文件中 来书写真正的用例代码
2. 代码文件的名字必须按照标识符的规则来书写(可以将代码的作用在文件的开头使用注释说明)

- 步骤

```
1. 导包 (unittest)
2. 自定义测试类
3. 在测试类中书写测试方法
4. 执行用例
```

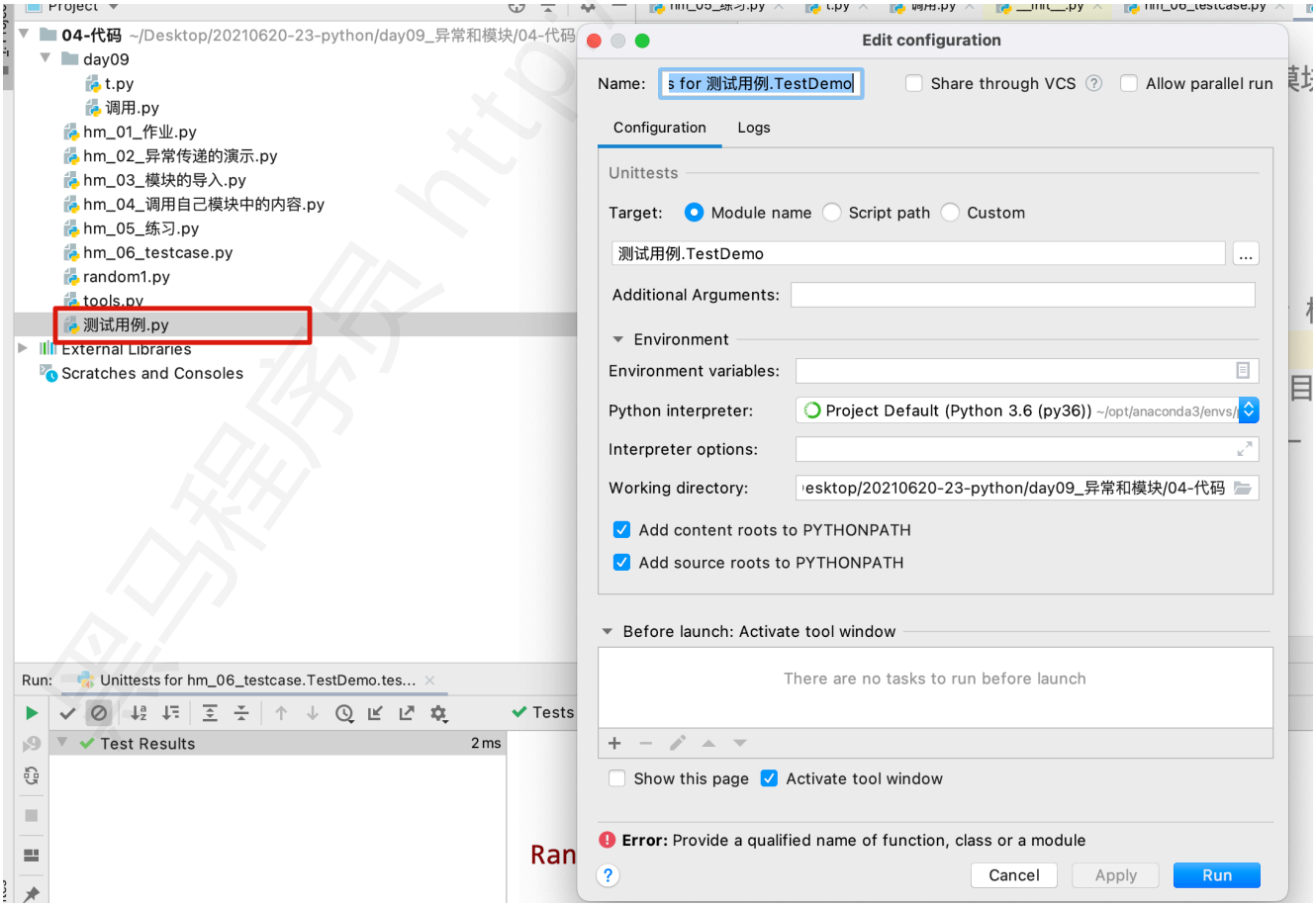
- 代码

```
"""
代码的目的: 学习 TestCase(测试用例)模块的书写方法
"""
# 1, 导包
import unittest


# 2, 自定义测试类, 需要继承 unittest 模块中的 TestCase 类即可
class TestDemo(unittest.TestCase):
    # 3, 书写测试方法, 即 用例代码. 目前没有真正的用例代码, 使用 print 代替
    # 书写要求, 测试方法 必须以 test_ 开头(本质是以 test 开头)
    def test_method1(self):
        print('测试方法 1')

    def test_method2(self):
        print('测试方法 2')
```

```
# 4，执行用例(方法)
# 4.1 将光标放在 类名的后边 运行，会执行类中的所有的测试方法
# 4.2 将光标放在 方法名的后边 运行，只执行当前的方法
```

问题一 代码文件的命名不规范

1．代码文件的名字以数字开头
2．代码文件名字中有空格
3．代码文件名字有中文
4．其他的特殊符号
（数字，字母，下划线组成，不能以数字开头）



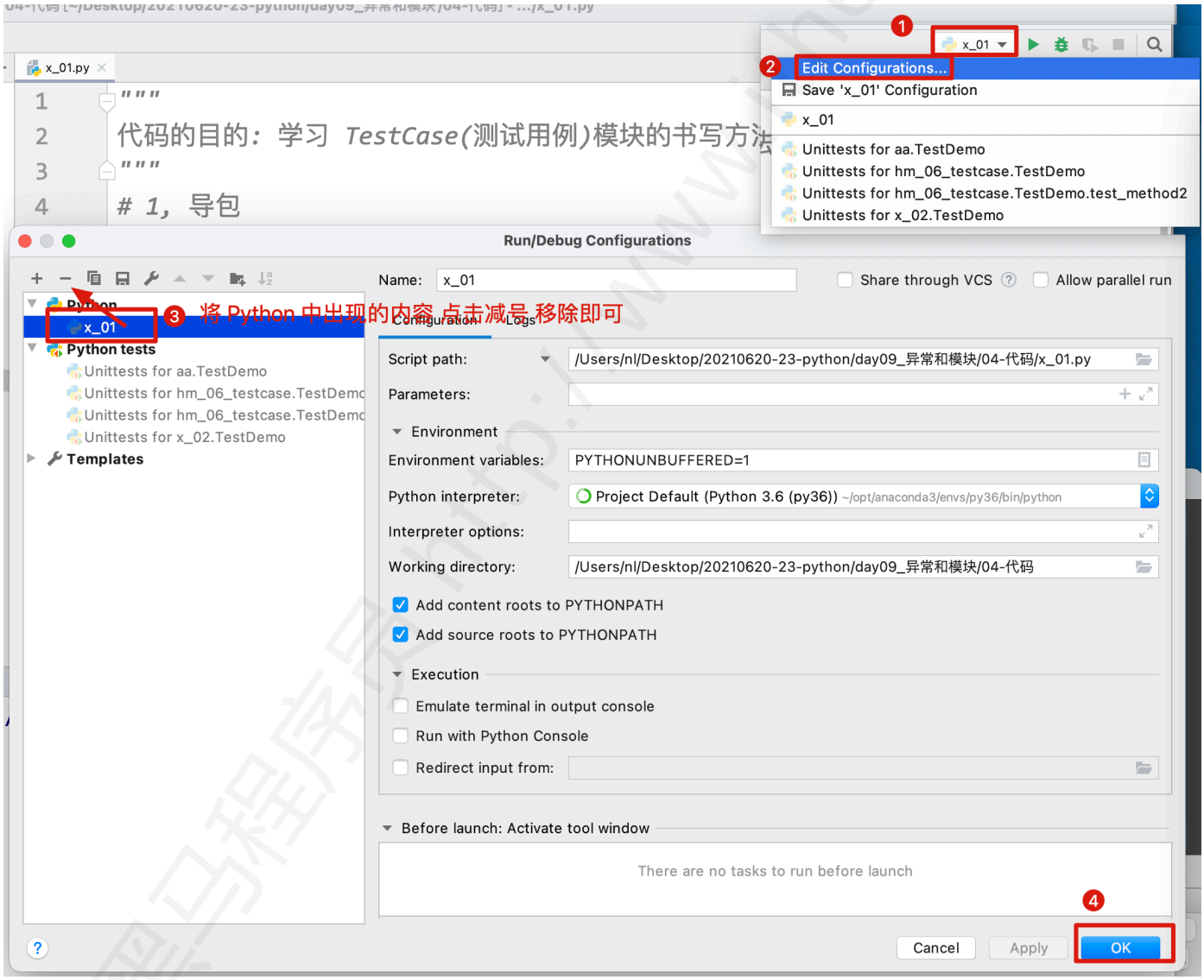
问题 2 代码运行没有结果

右键运行没有 unittests for 的提示，出现的问题

解决方案：

方案1. 重新新建一个代码文件，将写好的代码复制进去

方案2. 删除已有的运行方式



问题 3 没有找到用例

测试方法中不是以 test_ 开头的，或者单词写错了

TestSuite & TestRunner

TestSuite(测试套件): 管理 打包 组装 TestCase(测试用例)文件的
TestRunner(测试执行): 执行 TestSuite(套件)

- 步骤

  1. 导包(unittest)
  2. 实例化(创建对象)套件对象
  3. 使用套件对象添加用例方法
  4. 实例化运行对象
  5. 使用运行对象去执行套件对象

- 代码

  TestSuite(测试套件): 是用来管理多个 TestCase(测试用例) 的,
  先创建多个 TestCase(测试用例) 文件

```
"""
学习 TestSuite 和 TestRunner 的使用
"""
```

```python
# 1. 导包(unittest)
import unittest
from hm_07_testcase1 import TestDemo1
from hm_07_testcase2 import TestDemo2

# 2. 实例化(创建对象)套件对象,
suite = unittest.TestSuite()
# 3. 使用套件对象添加用例方法
# 方式一, 套件对象.addTest(测试类名('方法名'))   #
建议测试类名和方法名直接去复制,不要手写
suite.addTest(TestDemo1('test_method1'))
suite.addTest(TestDemo1('test_method2'))
suite.addTest(TestDemo2('test_method1'))
suite.addTest(TestDemo2('test_method2'))

# 4. 实例化运行对象
runner = unittest.TextTestRunner()
# 5. 使用运行对象去执行套件对象
# 运行对象.run(套件对象)
runner.run(suite)
```

```python
"""
学习 TestSuite 和 TestRunner 的使用
"""
# 1. 导包(unittest)
import unittest
```

```python
# 2. 实例化(创建对象)套件对象,
from hm_07_testcase1 import TestDemo1
from hm_07_testcase2 import TestDemo2

suite = unittest.TestSuite()
# 3. 使用套件对象添加用例方法
# 方式二 将一个测试类中的所有方法进行添加
# 套件对象.addTest(unittest.makeSuite(测试类名))
# 缺点: makeSuite()  不会提示
suite.addTest(unittest.makeSuite(TestDemo1))
suite.addTest(unittest.makeSuite(TestDemo2))

# 4. 实例化运行对象
runner = unittest.TextTestRunner()
# 5. 使用运行对象去执行套件对象
# 运行对象.run(套件对象)
runner.run(suite)
```

```
/Users/nl/opt/anaconda3/envs/py36/bin/python /Users/nl/Desktop/20210620-23-pyth
....        运行结果中 . 用例通过  F 用例不通过  E error 用例代码有问题
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
测试方法 1-1
测试方法 1-2
测试方法 2-1
测试方法 2-2

Process finished with exit code 0
```

练习

```
1. 在 tools 模块中定义 add 函数，对两个数字进行求和计算
2. 书写 TestCase 代码对 add() 进行测试
用例 1:  1, 2, 3
用例 2:  10, 20, 30
用例 3:  2, 3, 5
```

- 用例代码

```
"""案例练习"""
# 1,导包
import unittest
from tools import add

# 2, 自定义测试类


class TestAdd(unittest.TestCase):
    # 3. 书写测试方法，就是测试用例代码
    def test_method1(self):
        # 1, 2, 3 判断实际结果和预期结果是否相符
        if add(1, 2) == 3:
            print('测试通过')
        else:
```

```
            print('测试不通过')

    def test_method2(self):
        if add(10, 20) == 30:
            print('测试通过')
        else:
            print('测试不通过')

    def test_method3(self):
        # 1, 2, 3 判断实际结果和预期结果是否相符
        if add(2, 3) == 5:
            print('测试通过')
        else:
            print('测试不通过')
```

- 套件和执行的代码

```python
import unittest

# 实例化套件对象
from hm_08_test import TestAdd

suite = unittest.TestSuite()
# 添加测试方法
suite.addTest(unittest.makeSuite(TestAdd))
# 实例化执行对象
runner = unittest.TextTestRunner()
runner.run(suite)
```