执行方式一(执行所有的测试用例)

```
1.执行main()方法执行的特点:unittest.main()
    注意:
    1.是把所有的测试用例执行了一遍
    2.执行测试用例的顺序控制不了，（按照测试用例名（方法名）的字母顺序执行的）
```

执行方式二(内部方法添加测试用例)(重点)

```
通过内部函数添加测试用例
步骤:
1.生成测试套件(也叫测试集合)
    suitt = unittest.TestSuite()
2.把测试用例添加进测试集合,两种方式添加
    1.suitt.addTest(类名("用例"))
    2.suitt.addTests(map(类名,["用例1","用例2",...]))
3.生成测试结果对象,然后传递到run函数中
    re = unittest.TestResult()
    suitt.run(re)

#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import unittest

class Maker(unittest.TestCase):
    def test_01(self):
        print("test_01......")

    def test_02(self):
        print("test_02......")

    def test_03(self):
        print("test_03......")

    def test_04(self):
        print("test_04......")

    def test_05(self):
        print("test_05......")

if __name__=='__main__':
    # 1.
```

```
    # 生成测试套件(也叫测试集合)
    suitt = unittest.TestSuite()
    # 2.
    # 把测试用例添加进测试集合， 两种方式添加
    # 1.
    # suitt.addTest(Maker("test_03"))
    # 2.
    suitt.addTests(map(Maker, ["test_01", "test_03","test_05"]))
    # 3.
    # 生成测试结果对象， 然后传递到run函数中
    re = unittest.TestResult()
    suitt.run(re)
#需要使用python中的IDLE执行,或直接终端执行这个文件
```

执行方式二(通过模块方法添加测试用例)(重点)

```
通过模块方法添加测试用例
1.如果测试用例的数量比较大，使用testsuite自带的方法加用例到集合，很麻烦
可以unittest中提供的testloader模块，提供了好多帮我们把测试用例加载到测试集合中的方法
2.步骤:
    2.创建testloader的对象
    3.使用testloader的对象中的loadTestsFromName函数添加测试用例,这个api,返回测试套件
        1.可以添加整个模块的测试用例
        2.也可以添加一个单元测试类中的所有测试用例(只能添加一个单元测试类)
        3.也可以添加单元测试类中的某个测试用例(只能添加一个单元测试类中的一个测试用例)

    4.生成测试结果对象,然后传递到run函数中
        re = unittest.TestResult()
        suitt.run(re)


模块代码:
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import unittest

class Maker(unittest.TestCase):
    def test_01(self):
        print("test_01......")

    def test_02(self):
        print("test_02......")

    def test_03(self):
        print("test_03......")

```

```
    def test_04(self):
        print("test_04......")

    def test_05(self):
        print("test_05......")


class Maker2(unittest.TestCase):
    def test_01(self):
        print("test_01...Maker2...")

    def test_02(self):
        print("test_02....Maker2..")

    def test_03(self):
        print("test_03...Maker2...")

    def test_04(self):
        print("test_04...Maker2...")

    def test_05(self):
        print("test_05...Maker2...")

业务代码:
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import unittest
if __name__=='__main__':
    #创建loader对象
    loader=unittest.TestLoader()
    #参数为模块名,返回测试套件,执行一个模块中所有的用例
    suit=loader.loadTestsFromName("Maker2Tests")
    #执行一个模块中的某个单元测试类中的所有用例
    # suit=loader.loadTestsFromName("Maker2Tests.Maker")
    #执行一个模块中的某个单元测试类中的某个用例
    suit=loader.loadTestsFromName("Maker2Tests.Maker.test_05")

    # 生成测试结果对象, 然后传递到run函数中
    re = unittest.TestResult()
    suit.run(re)


```

执行方式二(通过路径方式添加测试用例)(重点)

```
使用unittest.defaultTestLoader对象的discover方法加载用例，可以将指定路径所有符合匹配规则（pattern）的文件中的单元测试用例一次性加载
```

```
第一个参数是一个目录，这个目录下可以有单元测试用例的文件（.py）
第二个参数是填文件名,可以通配
 suitt = unittest.defaultTestLoader.discover(r"./Maker/", pattern="unit*.py")
说明:
    1."unit*.py指的是以unit开头，以.py结尾的文件
    2..py中的单元测试用例要使用unittest框架写的测试用例

步骤:
    1.生成测试套件(也叫测试集合)
    2.使用discover方法批量添加
    3.生成测试结果对象,然后传递到run函数中
        re = unittest.TestResult()
        suitt.run(re)

#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import unittest

if __name__=='__main__':
    suit=unittest.defaultTestLoader.discover(r'./MyTestFile/',pattern="unit*.py")
    print(suit.countTestCases())
    # 生成测试结果对象，然后传递到run函数中
    re = unittest.TestResult()
    suit.run(re)
```

执行测试用例并生成执行报告

```
在前面测试用例、测试集合执行的时候都是用testsuite()的run()方法：suitt.run(result),如果要生成text文本形式的测试执行报告,可以使用TestRunner
将最后执行的:
 re = unittest.TestResult()
suitt.run(re)
改为:
with open(r"./re.txt", "w", encoding="utf-8") as f:
  runner = unittest.TextTestRunner(f, descriptions="单元测试报告执行", verbosity=5)
  runner.run(suitt)
函数说明:
f:文件描述符
descriptions:用来标记是否输出测试用例的描述信息。布尔类型,没什么用
```

```
verbosity参数可以控制输出的错误报告的详细程度，只有3个取值：
    0 (quiet): 只显示执行的用例的总数和全局的执行结果。
    1 (default): 默认值，显示执行的用例的总数和全局的执行结果，并对每个用例的执行结果（成功T或失败F）有个标注。(测试用例中如果有和预计不同,会出现F)
    2+ (verbose): 显示执行的用例的总数和全局的执行结果，并输出每个用例的详细的执行结果。


TextTestRunner()-->将结果能够以text文本形式展示的运行器

#使用TextTestRunner()运行器提供的run()方法运行测试集合

#如何产生一个文件流对象，如果打开一个文本文件，想着往里写入数据

#报告是以TextTestResult的形式展示的

#TextTestRunner是TestRunner的子类

#TextTestResult是TestResult的子类


如果没有生成1.txt,那么使用python的IDLE执行

```

断言

```
断言：一个自动化测试用例，测试步骤、测试的断言缺一不可
unittest中提供的断言方法有：
     assertEqual(a,b,msg=""):就是判断a和b是否相等，如果相等，则断言成功，如果不相等，会断言失败，并且输出msg消息
     assertNotEqual(a,b,msg="")：就是判断a和b是否不相等
     assertTrue(a):就是判断a是否为True这个bool值
     assertFalse(a):就是判断a是否为False这个bool值
     assertIs(a,b,msg=""):判断a和b的内存地址是否相等，如果相等则身份一致
     assertIsNot(a,b,msg):判断a和b的内存地址是否不相等，如果像等了，返回false，断言失败
     assertIsNone(a):判断对象a是不是空指针(没有指向堆内存中空间),如果是则断言成功
     assertIsNotNone(a):判断对象a是不是空指针，如果是，则断言失败
     assertIn(a,b):判断a是不是b的成员，如果是则断言成功
     assertNotIn((a,b):判断a是不是b的成员，如果不是则断言成功
     assertIsInstance(a,b):判断a是b的一个实例对象
     assertIsNotInstance(a,b):判断a不是b的一个实例对象


```

HTML格式展示执行报告(重点)

```
前面使用runner运行器是unittest自带的，效果不是很好，我们第三方开发的来用，可以以HTML格式展示结果。

HTMLTestRunner模块：就是一个第三方
1、下载该模块的python3版本

```

```
2、复制该文件到python的安装目录下/lib中或把HTMLTestRunner.py放到你的项目中
3、导包:from HTMLTestRunner import HTMLTestRunner

#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import unittest
from HTMLTestRunner import HTMLTestRunner
import time
if __name__=='__main__':
    suit = unittest.defaultTestLoader.discover(r'./MyTestFile/', pattern="unit*.py")

    filename="./"+time.strftime("%Y-%m-%d %H_%M_%S")+"res.html"
    with open(filename,"wb") as f:
        runner=HTMLTestRunner(f,verbosity=2,title="单元测试报告",description="第一次运行结果")
        runner.run(suit)
```

邮件的自动化

```
通过python我们可以自动发送报告给负责人
Python对SMTP支持有smtplib和email两个模块，email负责构造邮件，smtplib负责发送邮件
1.把html发送为正文
2.把html发送为附件

需要把发送邮件的邮箱的smtp/pop3的服务开通

发送为正文:
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
#构建邮件内容的
from email.mime.text import MIMEText
#构建邮件头部信息的
from email.header import Header
#构建发件人
from email.utils import formataddr
#创建发送邮件对象
import smtplib
```

```
#把测试报告发送为正文

#读取报告的内容
htmlmsg=open('2023-07-05 20_53_41res.html','rb').read()
#构建邮件正文
msg=MIMEText(htmlmsg,'html','utf-8')
#头部信息
msg['Subject']=Header("python邮件自动化",'utf-8')
#发件人信息
msg['From']=formataddr(['张三202305',"76754438@qq.com"])
#收件人
msg['To']="lzs8407@163.com"

#构建SMTP对象
smtp=smtplib.SMTP()
#连接发送邮件的邮箱服务器
smtp.connect("smtp.qq.com")
#登录
smtp.login("76754438@qq.com","uomuthspmlfgbhcg")
#发送
smtp.sendmail("76754438@qq.com","lzs8407@163.com",msg.as_string())
#退出
smtp.quit()


发送为附件
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
#构建邮件内容的
from email.mime.text import MIMEText
#构建邮件头部信息的
from email.header import Header
#构建发件人
from email.utils import formataddr
from email.mime.multipart import MIMEMultipart
#创建发送邮件对象
import smtplib
#发送附件的对象
MYmime=MIMEMultipart()
#读取报告的内容
htmlmsg=open('2023-07-05 20_53_41res.html','rb').read()
#构建邮件正文
msg=MIMEText(htmlmsg,'base64','utf-8')
msg['Content-Type']="application/octet-stream"
msg['Content-Disposition']='attachment;filename=1.html'
#头部信息
MYmime['Subject']=Header("python邮件自动化",'utf-8')
#发件人信息
```

```
MYmime['From']=formataddr(['张三20230888',"76754438@qq.com"])
#收件人
MYmime['To']="lzs8407@163.com"
MYmime.attach(msg)


#构建SMTP对象
smtp=smtplib.SMTP()
#连接发送邮件的邮箱服务器
smtp.connect("smtp.qq.com")
#登录
smtp.login("76754438@qq.com","uomuthspmlfgbhcg")
#发送
smtp.sendmail("76754438@qq.com","lzs8407@163.com",MYmime.as_string())
#退出
smtp.quit()


```

ddt

```
ddt是“Data-Driven Tests”的缩写，是unittest中实现数据驱动的主要方式之一，它主要包括如下的装饰器

@ddt
标记测试类，支持DDT数据驱动

@data
标记测试用例，传递参数

@unpack
当@data中的参数是元组、列表时，用于分割序列中的元素

@file_data
标记测试用例，传递文件，支持yaml和json文件

#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import unittest
import ddt

test_data=['hello','world']

@ddt.ddt#代表下面这个单元测试支持ddt驱动
class Testmaker(unittest.TestCase):

    @ddt.data(1,2)
```

```
    def test_ddt(self,v):#这个用例会执行2遍
        print("test_ddt")
        print(v)#1 2

    @ddt.data((1,2,3),[3,4,5])
    @ddt.unpack
    def test_ddt2(self,v1,v2,v3):#data参数中的序列有多少个数,那么这里的参数个数就必须是多少个
        print("test_ddt2")
        print(v1)
        print(v2)

    #代码为完成

if __name__=='__main__':
    unittest.main()


```

##