ddt

```
#!/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)

    @ddt.data(test_data)#参数是列表时
    @ddt.unpack
    def test_ddt3(self,v1,v2):
        print("test_ddt3")
        print(v1)
        print(v2)

    @ddt.file_data("mydata.yaml")#参数是文件名
    def test_ddt4(self,txt):#执行了4次,因为mydata.yaml中有4个数据
        print("test_ddt4")
        print(txt)

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

unittest使用ddt时,如果使用测试套件执行用例时,会出现添加不了测试用例的问题

```
问题: #注意:使用了ddt加载数据后,会改变测试用例的名字,导致添加进测试套件中出错
#!/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)

    @ddt.data(test_data)#参数是列表时
    @ddt.unpack
    def test_ddt3(self,v1,v2):
        print("test_ddt3")
        print(v1)
        print(v2)

    @ddt.file_data("mydata.yaml")#参数是文件名
    def test_ddt4(self,txt):#执行了4次,因为mydata.yaml中有4个数据
        print("test_ddt4")
        print(txt)

if __name__=='__main__':
    #注意:使用了ddt加载数据后,会改变测试用例的名字,导致添加进测试套件中出错
    # 生成测试套件(也叫测试集合)
    suitt = unittest.TestSuite()
    suitt.addTests(map(Testmaker, ["test_ddt", "test_ddt3", "test_ddt4"]))

    re = unittest.TestResult()
    suitt.run(re)
```

```
解决方法:
第一种方式,使用路径添加模块方式生成测试套件,这个方式不需要测试用例的名字
if __name__=='__main__':
    suit = unittest.defaultTestLoader.discover(r'./', pattern="03unit.py")
    re = unittest.TestResult()
    suit.run(re)

第二种方式:
if __name__=='__main__':
    suit = unittest.TestSuite()
    #获取ddt驱动之后的测试用例名字
    mylist=unittest.TestLoader().getTestCaseNames(Testmaker)
    print(mylist)
    suit.addTests(map(Testmaker,mylist))
    re = unittest.TestResult()
    suit.run(re)
```

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

#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
#场景:百度首页中点击新闻链接,测试新闻页面打开,并且正确,正确的话就是url是https://news.baidu.com/,
#而且浏览器有2个句柄

import unittest
from selenium import webdriver
import time
```

```
class Maker(unittest.TestCase):

    def setUp(self):
        self.dr=webdriver.Firefox()
        self.dr.get("https://www.baidu.com")
        time.sleep(2)

    def tearDown(self):
        time.sleep(3)
        self.dr.quit()

    def test_urlnew(self):
        self.dr.find_element_by_link_text('新闻').click()
        time.sleep(2)
        #获取浏览器的句柄个数
        mylist=self.dr.window_handles
        # 预期结果
        n=2
        #测试结果
        n2=len(mylist)
        self.assertEqual(n,n2,"测试不通过1")
        self.dr.switch_to.window(mylist[1])
        #预期url
        myurl="https://news.baidu.com/"
        #测试结果
        myurl2=self.dr.current_url
        self.assertEqual(myurl,myurl2,'测试不通过2')


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

POM模式介绍

```
虽然unittest单元测试框架在一定层度上解决了大量用例（100条、500条甚至几千条用例）所带来的用例管理和执行上的麻烦。

但是UI的元素属性变化带来的麻烦，就有心无力了，比如涉及到500条case的页面元素属性变化，现有脚本的定位出现问题，那岂不是要改500遍，那是不是会改到心碎。这种怎么解决呢？
使用POM模式
POM设计模式，即Page Object Model，这也是是目前最为经典的一种设计思想，用大白话说就是：将页面UI元素对象、逻辑、业务、数据等分离开来，使得代码逻辑更加清晰，复用性，可维护性更高的一种方法。

POM模式的优点
让UI自动化更早介入项目中，可项目开发完再进行元素定位的适配与调试
POM 将页面元素定位和业务操作流程分开，分离了测试对象和测试脚本
如果UI页面元素更改，测试脚本不需要更改，只需要更改页面对象中的某些代码就可以
```

POM能让我们的测试代码变得可读性更好，高可维护性，高复用性，
可多人共同维护开发脚本，利于团队协作

ddt读取yaml文件内容
1.每组数据用‘-’隔开，一个-就是一组数据
2.读取yaml文件必须是test开头的函数

要独立创建项目，不然导入模块会出问题

项目下有4个目录:
base目录,里面的文件都是写selenium相关的代码,如元素定位,操作元素等
data目录,里面是存储数据的文件
cases目录,里面是业务代码,就是在单元测试类中写测试用例
page_Object目录:这里是封装业务代码的

pom模式是把selenium的api,数据,页面元素的属性,业务这些分离开来,
如果seleniium的api改变了,只需修改base里的代码
如果数据变了,只需修改data里的内容
如果页面元素的属性变了,只需修改page_Object里的代码
如果业务代码变了,只需修改cases里的内容

base:
```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
#页面操作或定位元素等代码都写到这
from selenium import webdriver
class BasePage():
    def __init__(self,url):
        self.dr=webdriver.Firefox()
        self.dr.get(url)

    #*lcator是(By.ID,"kw")
    def find_element(self,*lcator):
        e=self.dr.find_element(lcator)
        return e

    def mysend_keys(self,lcator,value,is_clear=True):
        e=self.dr.find_element(*lcator)
        if is_clear:
            e.clear()
            e.send_keys(value)
        else:
            e.send_keys(value)


page_Object目录:
#!/usr/bin/env python
# -*- coding:utf-8 -*-
```

```
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
from base.base_page import BasePage
from selenium.webdriver.common.by import By
class Seach_page(BasePage):
    def __init__(self,url):
        BasePage.__init__(self,url)
        self.name_locat=(By.ID,'kw')

    def input_name(self,value):
        self.mysend_keys(self.name_locat,value)

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

from page_Object.seach_page import Seach_page
import unittest
import ddt

@ddt.ddt
class Maker(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.res=Seach_page("https://www.baidu.com")

    @ddt.file_data("../data/mydata.yaml")
    def test_res(self,txt):
        self.res.input_name(txt)

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

pytest的介绍

```
pytest是Python的一款单元测试框架,一种全新的框架思维来管理和规范我们的测试脚本，从而实现高类聚低耦合的理念

优势:
1.通用性
2.丰富的第三方库
3.入门快速
4.精美的报告
5.定制性强
```

pytest的安装

```
安装方式一：
1.pycharm上创建一个文件，输入import pytest
2.鼠标点击pytest,出现小灯泡，点击小灯泡进行下载

安装方式二：
1.pycharm上终端，输入:pip install -U pytest

安装方式三：
1.pycharm上的菜单-文件-设置-项目xxx-project Interpreter中下载

```

pytest的约束

```
1.所有的单测文件名都需要满足test_*.py格式或*_test.py格式。
2.在单测文件中，测试类以Test开头，并且不能带有 init 方法(注意：定义class时，需要以Test开头，不然pytest是不会去运行该class的)
3.在单测类中，可以包含一个或多个test_开头的函数。
此时，在执行pytest命令时，会自动从当前目录及子目录中寻找符合上述约束的测试函数来执行。


```

pytest的helloworld

```
目的测试一个加法
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import pytest

def test_01():
    print("hello pytest")

    a=10+20
    b=30
    assert a==b

if __name__=='__main__':
    #如果使用下面的方式,文件名不受规则限制
    pytest.main(["01hellopytest.py","-s"])

```

pytest的命令

```
import pytest
# #执行别的文件中所有用例
# pytest.main(["./01hello_pytest.py","-s"])
# #查看pytest的版本
# pytest.main(["--version"])
# #查看内置函数参数
# pytest.main(["--fixtures"])
#查看帮助信息
# pytest.main(['--help'])
```

pytest的执行用例

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import pytest
#1.执行某个目录中所有的用例
# pytest.main(["./mycasetest/"])
#2.执行某个目录中某个模块的所有用例
# pytest.main(["./mycasetest/test_02.py"])
#3.执行某个目录中某个模块的某个用例
# pytest.main(["./mycasetest/test_02.py::test_01"])
#4.执行某个目录中某个模块的某几个用例,用例的上面需要标记@pytest.mark.slow,slow自己取的名字
pytest.main(["./mycasetest/test_02.py::test_01","-m","slow"])
代码未完成

```