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"])
#执行某个模块中某个单元测试类中的某个测试用例
# pytest.main(["./mycasetest/test_03.py::Test_maker::test_m2","-s"])
#执行某个目录中的所有用例 但有执行到错误就停止
# pytest.main(['./mycasetest/','-x'])
# 执行某个目录中的所有用例 ,指定出错的个数,如果达到个数,就停止后面的执行
# pytest.main(['./mycasetest/','--maxfail=2',"-s"])
#通过匹配关键字来执行用例,注意,如果是test_0,其实包含到了_后面的字母
# pytest.main(['./mycasetest/','-k','test_1','-s'])
#通过匹配来实现文件中有包含maker的类名的单元测试来
# pytest.main(['./mycasetest/','-k','maker','-s'])
#不执行某个单元测试类中的某个用例
# pytest.main(['./mycasetest/','-k','maker and not test_m2','-s'])
#不执行目录中的一些用例
pytest.main(['./mycasetest/','-s','-k','not test_1'])

```

pytest自定义fixture

```
fixture:即测试用例执行的环境准备和清理，在unittest中指setup/teardown
fixture:主要的目的是为了提供中可靠和可重复性的手段去运行那些最基本的测试内容。

fixtue的作用范围
scope参数，控制fixture函数的作用范围
scope = "function" ，默认范围，针对测试函数(测试方法)，测试用例执行前会执行该函数的yield前面部分,相当于setUp,测试用例执行完成后去执行yield后部分,相当于teraDwon


scope = "class" ， 测试类，类当中所有的测试用例执行前会执行该函数的yield前面部分,相当于setUpClass,测试用例执行完成后去执行yield后部分,相当于teraDwonClass

```

```
scope = "module" ， 测试模块，一个模块当中的所有的测试用例执行前调用函数的yield前面部分,相当于整个模块的setUpClass,模块中所有的用例完毕后会执行yield后部分,相当于整个模块的teraDwonClass

scope = "session"， 测试会话， 当调用别的文件的fixture时,会调用
1.fixture的基本使用:
手动调用fixture修饰的函数
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import pytest

@pytest.fixture(scope='function')
def mysetup():
    print("mysetup")


class TestDome():
    def test_01(self,mysetup):
        print("test_01")
        assert 1

    def test_02(self,mysetup):
        print("test_02")
        assert 1

if __name__=='__main__':
    pytest.main(['02fixture的使用.py','-s'])




2.自动调用
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import pytest

@pytest.fixture(scope='function',autouse=True)
def mysetup():
    print("mysetup")


class TestDome():
```

```
    def test_01(self):
        print("test_01")
        assert 1

    def test_02(self):
        print("test_02")
        assert 1

if __name__=='__main__':
    pytest.main(['02fixture的使用.py','-s'])


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

import pytest

@pytest.fixture(scope='class',autouse=True)
def mysetupclass():#相当于setUpClass,只执行一次
    print("mysetupclass")
    assert 1

class TestDome():
    def test_01(self):
        print("test_01")
        assert 1

    def test_02(self):
        print("test_02")
        assert 1

if __name__=='__main__':
    pytest.main(['03fixture的使用2.py','-s'])

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

import pytest

@pytest.fixture(scope='function',autouse=True)
def mysetup():
```

```
    print("mysetup")
    yield
    print("teardown...")


class TestDome():
    def test_01(self):
        print("test_01")
        assert 1

    def test_02(self):
        print("test_02")
        assert 1

if __name__=='__main__':
    pytest.main(['04fixture的使用3.py','-s'])

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

import pytest

@pytest.fixture(scope='class',autouse=True)
def mysetupclass():#相当于setUpClass,只执行一次
    print("mysetupclass")
    yield
    print("teardownclass")
    assert 1

class TestDome():
    def test_01(self):
        print("test_01")
        assert 1

    def test_02(self):
        print("test_02")
        assert 1

if __name__=='__main__':
    pytest.main(['05fixture的使用4.py','-s'])


5测试模块
test_maker.py内容
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
```

```
#Author:
#CreatDate:
#Version:
#====#====#====#====


import pytest

@pytest.fixture(scope='module',autouse=True)
def fix():
    print("fix setupclass")
    yield
    print("fix teardownclass")

def test_case1():
    print("test_case1")

def test_case2():
    print("test_case2")

06fixture的使用5.py
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

import pytest

pytest.main(['test_maker.py','-s'])
```

测试用例中使用自定义fixture的返回值

```
手动调用
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
#手动调用
import pytest

@pytest.fixture(scope='function')
def mytest():
    print("mytest")
    return "mytest返回数据"

def test_01(mytest):
```

```
    print("test_01")
    print(mytest)

if __name__=='__main__':
    pytest.main(['07fixture的返回值1.py','-s'])



自动调用
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
#手动调用
import pytest

@pytest.fixture(scope='function',autouse=True)
def mytest():
    print("mytest")
    return "mytest返回数据"

def test_01(mytest):
    print("test_01")
    print(mytest)

if __name__=='__main__':
    pytest.main(['07fixture的返回值1.py','-s'])


```

fixure 中获取params 传入的数据 实现数据驱动

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import pytest
def my_data():
    return ['1111','2222','3333']

@pytest.fixture(scope="function",params=my_data(),autouse=True)
def mytest(request):#固定写request
    print("mytest")
    print(request.param)

def test_01():#数据有三个,所以调用3次.如果test_01想要数据,那么就mytest返回,然后test_01使用
```

```
    print("test_01")

if __name__=='__main__':
    pytest.main(['08fixture实现数据驱动.py','-s'])

```

测试用例中使用 fixture的yield后的数据

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import pytest
def my_data():
    return ['1111','2222','3333']

@pytest.fixture(scope="function",params=my_data(),autouse=True)
def mytest(request):#固定写request
    print("mytest")
    yield  request.param
    print("end mytest")

def test_01(mytest):#数据有三个,所以调用3次.如果test_01想要数据,那么就mytest返回,然后test_01使用
    print("test_01")
    print(mytest)

if __name__=='__main__':
    pytest.main(['09fixture的yield后的数据.py','-s'])

```

ids 与params参数结合使用，给参数起别名

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import pytest
def my_data():
    return ['1111','2222','3333']

@pytest.fixture(scope="function",params=my_data(),autouse=True,ids=['a','b','c'])
def mytest(request):#固定写request
    print("mytest")

```

```
    yield  request.param
    print("end mytest")

def test_01(mytest):#数据有三个,所以调用3次.如果test_01想要数据,那么就mytest返回,然后test_01使用
    print("test_01")
    print(mytest)

if __name__=='__main__':
    pytest.main(['10fixture的ids给参数取别名.py','-s','-v'])

```

name 参数 给fixture 起别名

```
需要注意的是一旦给fixture 起了别名后，原来被@pytest.fixture 标记过的方法名称就失效了

#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import pytest
def my_data():
    return ['1111','2222','3333']

@pytest.fixture(scope="function",params=my_data(),autouse=True,ids=['a','b','c'],name='run_sql')
def mytest(request):#固定写request
    print("mytest")
    yield  request.param
    print("end mytest")

def test_01(run_sql):#因为name="run_sql",等于给上面被fixture修饰的函数取别名叫run_sql
    print("test_01")
    print(run_sql)

if __name__=='__main__':
    pytest.main(['11name参数给fixture起别名.py','-s','-v'])

```

fixture 结合 conftest.py 文件使用

```
conftest.py它是写fixture的配置内容的专门模块,名字是固定的
在conftest.py模块中所以方法调用时不需要导包

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

```
#Author:
#CreatDate:
#Version:
#====#====#====#====
import pytest

class TestMaker():
    def test1(self):
        print("test1 run....")

    def test2(self):
        print("test2 run....")

    def setup_class(self):
        print("setup_class")

    def teardown_class(self):
        print("teardown_class")

    def setup(self):
        print("setup")

    def teardown(self):
        print("teardown")

if __name__=='__main__':
    pytest.main(['12fixture结合conftest文件使用.py','-s'])

'''
会话级别 setup
类级别 setup
setup_class
函数级别 setup
setup
test1 run....
.teardown
函数级别 teardown
函数级别 setup
setup
test2 run....
.teardown
函数级别 teardown
teardown_class
类级别 teardown
会话级别 teardown
'''

```

pytest执行过程

```
1.查询根目录下的conftest.py文件
2.查询根目录下的pytest.ini文件，找到测试用例的位置
3.查询测试用例目录下的conftest.py文件
4.查询测试用例的py文件中是否有setup,teardown,setup_class,teardown_class
5.再根据pytest.ini文件的测试用例规则去查找用例并执行
```

pytest参数化

```
熟悉unittest单元测试框架的小伙伴知道，使用ddt进行数据驱动测试，那么身为功能更加强大且更加灵活的Pytest框架怎么可能没有数据驱动的概念呢？Pytest使用@pytest.mark.parametrize装饰器来实现数据驱动测试的，也就是常说的参数化。

parametrize语法
parametrize(self,argnames, argvalues, indirect=False, ids=None, scope=None)
参数说明:
argnames：参数名。

argvalues：参数对应值，类型必须为list。如果只有一个参数，里面则是值的列表：

如：@pytest.mark.parametrize("username", ["yy", "yy2", "yy3"])。如果有多个参数，则需要用元组来存放值，一个元组对应一组参数的值，如：@pytest.mark.parametrize("name,pwd", [("yy1", "123"), ("yy2", "123"), ("yy3", "123")])。


indirect：如果设置成True，则把传进来的参数当函数执行，而不是一个参数。

ids：用例的ID，传一个字符串列表，用来标识每一个测试用例，自定义测试数据结果，增加可读性。

1.单个数据
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import pytest
data=["小明","小花"]
@pytest.mark.parametrize("name",data)
def test_dome(name):
    print("test_dome")
    print(name)


if __name__=='__main__':
    pytest.main(['13pytest参数化.py','-s'])


2.一组数据
a.列表嵌套字典
#!/usr/bin/env python
```

```
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import pytest
data=[
    {"username":"admin1","passwd":"123"},
    {"username":"admin2","passwd":"321"}
]
@pytest.mark.parametrize("name",data)
def test_dome(name):
    print("test_dome")
    print(name)


if __name__=='__main__':
    pytest.main(['13pytest参数化.py','-s'])

代码为完成
```