# Pytest单元测试框架

**一、自动化测试框架为什么要引入单元测试框架**

在自动化测试的框架中引入单元测试框架，可以更好的维护和管理测试用例，而且通过单元测试框架还可以很方便的对测试结果进行统计和生成测试报告

**二、pytest优点**

* 允许直接使用assert进行断言，而不需要使用self.assert\*;
* 可以自动寻找单测文件、类和函数;
* Modular fixtures可以用于管理小型或参数化的测试信息;
* 与unittest和nose单测框架兼容;
* 兼容性较好，支持Python 2.7，Python 3.4+。
* 丰富的插件支持，共计有超过315个插件支持;

**三、安装**

pip install pytest

**四、创建测试用例**

1. 测试用例文件及测试方法、测试类格式要求：
2. 所有的单测文件名都需要满足test\_\*.py格式或\*\_test.py格式。
3. 在单测文件中，可以包含test\_开头的函数，也可以包含Test开头的类。
4. 在单测类中，可以包含一个或多个test\_开头的函数
5. 测试函数

def func(x):

return x + 1

def test\_answer():

assert func(3) == 5

1. 测试类，测试类中可以包含多个测试类方法，方便组织对有关联、测试功能相关的 测试用例进行分组

class TestClass(object):

def test\_one(self):

x = "this"

assert 'h' in x

def test\_two(self):

x = "hello"

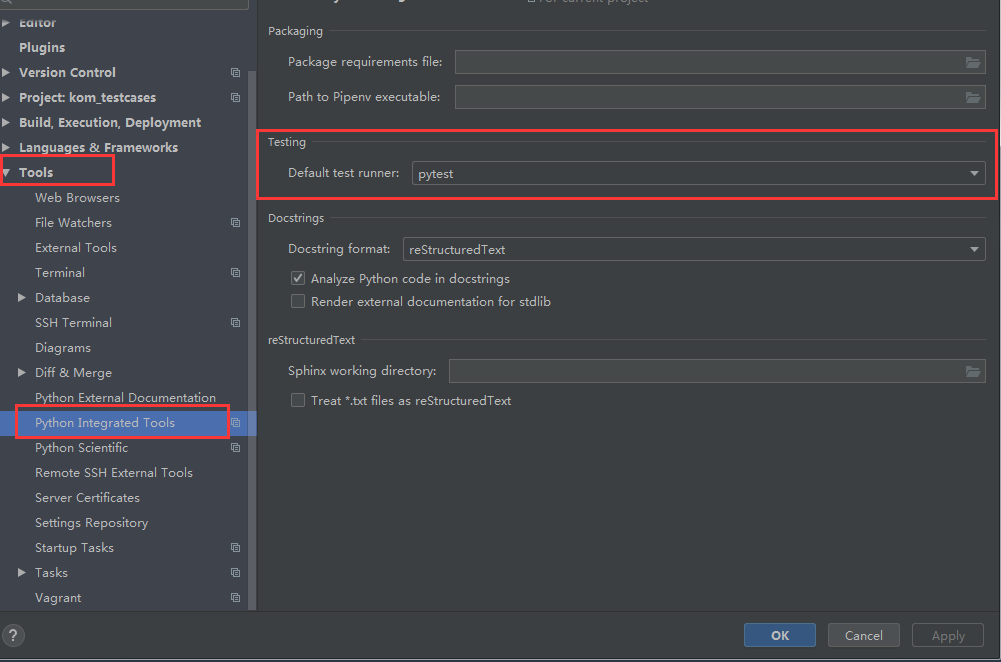
assert hasattr(x, 'check')

1. **Assert断言**
2. 断言即期望函数返回值或者变量的值为某个确定的值，如果值与期望不相等，则抛出 断言异常
3. Pytest可以使用python的标准断言assert语句，

**六、执行用例**

1. 设置pycharm运行pytest用例：

File --> Settings --> Tools -- > Python Intergated Tools，Testing设置为pytest



1. 运行单个测试用例文件：

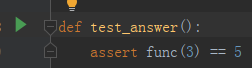
* Pycharm中运行：打开测试文件，点击右键，选择Run ‘pytest for test\_xxxxxxx’



* 命令执行：在测试用例文件目录下执行pytest test\_xxx.py

1. 执行单个测试用例：

* Pycharm中运行：点击测试用例旁边的绿色小箭头



* 命令执行：在测试目录下执行pytest test\_demo.py::test\_answer

1. 执行所有测试用例：

* Pycharm中运行：project视图下选择顶层目录点击右键选择



* 命令执行：在顶层测试目录下执行pytest命令

**七、pytest fixture**

1）、什么是fixture

fixture是被@pytest.fixture标记的函数。Fixture的目的是为了提供一个测试用例能可靠且稳定执行的固定基线

2）、如何定义fixture

@pytest.fixture()

def before():

print '\nbefore each test'

3）、fixture的作用范围

fixture里面有个scope参数可以控制fixture的作用范围：session>module>class>function

-function：每一个函数或方法都会调用

-class：每一个类调用一次，一个类中可以有多个方法

-module：每一个.py文件调用一次，该文件内又有多个function和class

-session：是多个文件调用一次，可以跨.py文件调用，每个.py文件就是module

1. 、fixture的使用
2. Fixture作为测试函数的参数
3. @pytest.mark.usefixtures("before")
4. 通过用autouse自动执行
5. 、参数化fixture

在定义fixture的装饰器@pytest.fixture中添加params参数，并通过pytest内置的request fixture获取参数的值，params参数列表中有多少个值，调用fixture的测试用例就会重复执行多少次，

@pytest.fixture(params=[1,2,3,4])

def param(request):

print('params value:%s'%request.param)

return request.param

1. 、fixture实现setup与teardown

在测试用例执行过程中，设置测试执行环境的前置条件设置与测试执行后的测试过程中产生的数据清理操作必不可少，在传统的unitest单元测试框架中，通过setUp与teardown来用于测试用例执行前的初始化工作与测试用例执行后的善后工作，pytest单元测试框架没有setUp与teardown，通过在fixture中用yield实现teardown功能，用autouse实现setup功能

**八、Conftest.py**

Fixture可以与测试用例文件分开，fixture可以放在conftest.py文件中，在conftest.py同级及同级以下目录的测试模块不需要导入即可以使用在conftest.py定义的fixture

**九、参数化测试用例**

1.使用@pytest.mark.parametrize进行参数

a.单次使用参数

@pytest.mark.parametrize("test\_input,expected", [("3+5", 8), ("2+4", 6), ("6\*9", 42)])

def test\_eval(test\_input, expected):

assert eval(test\_input) == expected

a..多次使用参数化

@pytest.mark.parametrize("x", [0, 1])

@pytest.mark.parametrize("y", [2, 3])

def test\_foo(x, y):

pass

1. 使用pytest\_generate\_tests内置钩子函数

（暂时不深入）

1. 运用indirect对fixture传参数

在@pytest.mark.parametrize中，可以应用indirect关键字将参数值传递给同名fixture， 参数的值在fixture中通过request内置fixture获取(request.fixture)并处理后传递给测试函数中的参数名

例：

import pytest

@pytest.fixture(scope='function')

def x(request):

return request.param \* 3

@pytest.fixture(scope='function')

def y(request):

return request.param \* 2

@pytest.mark.parametrize('x, y', [('a', 'b')], indirect=['x'])

def test\_indirect(x,y):

assert x == 'aaa'

assert y == 'b'

上例中，通过indirect关键字，可以将测试函数的参数x的值通过同名fixture处理后并返回

1. **对测试用例进行标记**

某些时候需要只运行部分的测试用例，比如在冒烟测试中，只需要运行冒烟测试用例，因此在编写测试用例时就需要对测试用例进行标记，以便在运行测试用例时可以根据标记只运行需要的测试用例。Pytest中可以使用@pytest.mark对用例进行标记，单个测试函数、测试类、测试模块多次标记

1).标记示例

例：

对单个测试函数进行标记

@pytest.mark.webtest

def test\_send\_http():

pass

对测试类进行标记

@pytest.mark.webtest

class TestClass(object):

def test\_startup(self):

Pass

class TestClass(object):

pytestmark = [pytest.mark.webtest, pytest.mark.slowtest]

对整个测试模块进行标记

import pytest

pytestmark = pytest.mark.webtest

pytestmark = [pytest.mark.webtest, pytest.mark.slowtest]

1. 根据标记选择运行用例
2. pytest -m webtest 只运行带webtest的标记用例
3. pytest -m "not webtest" 运行不带webtest标记的所有用例
4. 流程用例

对于某些流程或者场景测试，测试动作由一系列测试步骤组成，如果其中任何一步测试执行失败，则整个用例则应失败，在编写测试用例过程中，可以将流程用例的测试步骤放到一个测试类中，并对测试类进行流程标记

# content of conftest.py

def pytest\_runtest\_makereport(item, call):

if "incremental" in item.keywords:

if call.excinfo is not None:

parent = item.parent

parent.\_previousfailed = item

def pytest\_runtest\_setup(item):

if "incremental" in item.keywords:

previousfailed = getattr(item.parent, "\_previousfailed", None)

if previousfailed is not None:

pytest.xfail("previous test failed (%s)" % previousfailed.name)

# content of test\_step.py

import pytest

@pytest.mark.incremental

class TestUserHandling(object):

def test\_login(self):

pass

def test\_modification(self):

assert 0

def test\_deletion(self):

pass