### 1.1.16 功能：请求一个独特的临时目录

对于功能测试，经常要求创建一些文件并将其传递给应用对象，pytest提供内置属性，允许提供任意的资源，比如一个唯一的临时目录。

#content of file test\_tmpdirs.py

def **test\_needsfiles**(tmpdir):

print (tmpdir)

assert 0

我们在函数列表中列举了一个参数tmpdir,pytest会查找并调用fixture工厂，在测试函数调用之前，创建资源。让我们来运行它：

lzhy@lzhy-VirtualBox:~/Test/test2$ py.test -q test\_tmpdirs.py

F

=============================================================================================== FAILURES ================================================================================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_needsfiles \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

tmpdir = local('/tmp/pytest-of-lzhy/pytest-1/test\_needsfiles0')

def test\_needsfiles(tmpdir):

print (tmpdir)

> assert 0

E assert 0

test\_tmpdirs.py:3: AssertionError

----------------------------------------------------------------------------------------- Captured stdout call ------------------------------------------------------------------------------------------

/tmp/pytest-of-lzhy/pytest-1/test\_needsfiles0

1 failed in 0.02 seconds

在测试函数被调用之前，一个唯一的每测试被调用临时目录被创建了。

你可以查看有哪些内置属性，通过键入：

py.test --fixtures

## 1.2 使用和调用

### 1.2.1 调用pytest通过python -m pytest

你可以执行测试通过python命令行：

python -m pytest [...] #例：python -m pytest test\_class.py

以上命令等同于直接执行：py.test [...]

### 1.2.2 帮助

py.test --version # shows where pytest was imported from

py.test --fixtures # show available builtin function arguments

py.test -h | --help # show help on command line and config file options

### 1.2.3 停止测试在第N次失败后

py.test -x # stop after first failure

py.test --maxfail=2 # stop after two failures

### 1.2.4 指定和选择测试

py.test test\_mod.py # run tests in module

py.test somepath # run all tests below somepath

py.test -k stringexpr # only run tests with names that match the

# "string expression", e.g. "MyClass and not method"

# will select TestMyClass.test\_something

# but not TestMyClass.test\_method\_simple

py.test test\_mod.py::test\_func # only run tests that match the "node ID",

# e.g "test\_mod.py::test\_func" will select

# only test\_func in test\_mod.py

py.test test\_mod.py::TestClass::test\_method # run a single method in

# a single class

导入’pkt’，利用其系统位置进行查找和测试：

py.test --pyargs pkg # run all tests found below directory of pypkg

### 1.2.5 修改traceback打印

py.test --showlocals # show local variables in tracebacks

py.test -l # show local variables (shortcut)

py.test --tb=auto # (default) 'long' tracebacks for the first and last

# entry, but 'short' style for the other entries

py.test --tb=long # exhaustive, informative traceback formatting

py.test --tb=short # shorter traceback format

py.test --tb=line # only one line per failure

py.test --tb=native # Python standard library formatting

py.test --tb=no # no traceback at all

### 1.2.6 测试失败扔给PDB

### 1.2.7 设置断点

### 1.2.8 分析测试执行时间

获取执行最慢的10个测试执行时间：

py.test --durations=10

### 1.2.9 生成JUnitXML格式文件

生成可以被Hudson或其他集成测试服务器读取的结果文件：

py.test --junitxml=path #py.test --junitxml=/home/lzhy/Test/result.xml

### 1.2.10 生成测试结果文件

生成文本可读的测试结果文件：

py.test --resultlog=path

### 1.2.11 发送测试报告到在线pastebin服务

每个测试失败生成一个URL：

py.test --pastebin=failed

提交测试信息到远程paste服务并提供一个URL。你可以选择像平常那样测试，或者添加-x 选择一个特殊的失败测试。

完整测试日志生成url：

py.test --pastebin=all

### 1.2.12 禁用插件

禁止调用指定的插件，使用-p 选项和no前缀：

例如禁用doctest插件，

py.test -p no:doctest

### 1.2.13 从Python代码中调用Pytest

Pytest 2.0版本，你可以在python代码中直接调用pytest：

pytest.main()

这等同于通过命令行调用py.test。它不会抛出SystemExit，但是会返回退出码。你也可以传递选项和参数：

pytest.main(['-x', 'mytestdir'])或者通过字符串传递：pytest.main("-x mytestdir")

你也可以在pytest.main中指定额外的plugin。

import pytest

class **MyPlugin**:

def **pytest\_sessionfinish**(*self*):

print(*"\*\*\* test run reporting finishing"*)

pytest.main(*"-qq"*, plugins=[MyPlugin()])

## 1.3 良好的集成用用

### 1.3.1 pytest测试约定

Pytest实现以下标准测试约定：

* 如果不指定参数，测试收集从testpaths（如果配置）或当前路径。另外,命令行参数可以用在任何目录、文件名称或节点id。
* 递归目录，除非它是无递归的目录。
* test\_\*.py或者\*\_test.py文件，通过测试包名导入。
* 测试类要加Test前缀（不要有\_\_init\_\_方法）
* 测试函数和方法前加test\_.

在Python模块,pytest也发现unittest测试和TestCase用例子类化技术。

### 1.3.2 选择一个测试布局导入规则

Pytest支持两种常见的测试布局：

* 测试目录与真实的代码目录分开：

setup.py # your setuptools Python package metadata

mypkg/

\_\_init\_\_.py

appmodule.py

tests/

test\_app.py

...

* 内置测试目录到应用代码目录，如果测试与应用有直接联系，或者测试代码想随应用一起发布代码：

setup.py # your setuptools Python package metadata

mypkg/

\_\_init\_\_.py

appmodule.py

...

test/

test\_app.py

...

针对这两种形式重要提示：

* 确保’mypkg’包可导入，例如通过键入：

pip install -e . # install package using setup.py in editable mode

* 避免在测试目录下有”\_\_init\_\_.py”文件，因为这样当安装mypkg包时，测试用例很容易就被执行，因此独立于安装包，不管测试目录下有没有测试用例。
* 对于tests与应用代码同目录情况，\_\_init\_\_.py会与test同目录，可使用py.test --pyargs mypkg调用pytest检测应用代码被安装在哪，测试用例在哪。

py.test tests/test\_app.py # for external test dirs

py.test mypkg/test/test\_app.py # for inlined test dirs

py.test mypkg # run tests in all below test directories

py.test # run all tests below current dir

• determine basedir: this is the first “upward” (towards the root) directory not containing an \_\_init\_\_.py.

If e.g. both a and b contain an \_\_init\_\_.py file then the parent directory of a will become the basedir.

• perform sys.path.insert(0, basedir) to make the test module importable under the fully qualifiedimport name.

• import a.b.test\_module where the path is determined by converting path separators / into ”.” characters.This means you must follow the convention of having directory and file names map directly to the import

names.

### 1.3.3 tox

2

## 2 用法

py.test 框架会在它测试的项目中寻找 conftest.py 文件，然后在这个文件中寻找针对整个项目的选项，比如是否检测并运行 doctest 以及应该使用哪种模式检测测试文件和函数

py.test 的插件机制很丰富灵活.  
单个目录下的conftest.py, 全局conftest.py, 默认的fixture, 第三方库(比如pytest-random), 自定义插件等等.  
默认fixture全局有效, conftest.py在其作用域内有效, 第三方库和自定义插件可以自行设定 -p 或 赋值 pytest\_plugins.  
某些case不想用插件了, 可以用 -p no:plugin-name 反注册.  
hook 也有不少.  
比如添加addoption, parse cmd  
自定义报错.  
自定义ini,  
不胜枚举.  
从已知的pytest开源插件源代码随便翻看看, 大部分都或多或少使用了这些hook.

### 2.1 失败用例报告演示

### 2.2 基本模式和例子

### 2.2.1 传递不同的值给测试函数，取决于命令行参数

## 2.3 参数化测试

Pytest允许轻松地用参数表示测试函数。

### 2.3.1 生成参数组合，根据命令行

假设我们想要执行一个测试使用不同的计算参数，参数的范围由命令行决定

### PyTest fixtures：显示、模块化、可伸缩的

### Fixture 终结/执行teardown代码

Pytest支持执行指定的终结代码当fixture将要退出scope的时候。通过在fixture中接受一个请求对象，调用一次或多次request.addfinalizer。

*# content of conftest.py*

**import** **smtplibimport** **pytest**

**@pytest.fixture**(scope="module")**def** smtp(request):

smtp = smtplib.SMTP("smtp.gmail.com")

**def** fin():

**print** ("teardown smtp")

smtp.close()

request.addfinalizer(fin)

**return** smtp *# provide the fixture value*

### Fixtures内省请求的测试内容

Fixture函数接受请求对象内省“请求”测试函数、类、模块内容。

*# content of conftest.py***import** **pytestimport** **smtplib**

**@pytest.fixture**(scope="module")**def** smtp(request):

server = getattr(request.module, "smtpserver", "smtp.gmail.com")

smtp = smtplib.SMTP(server)

**def** fin():

**print** ("finalizing *%s* (*%s*)" % (smtp, server))

smtp.close()

request.addfinalizer(fin)

**return** smtp

我们使用request.module的模块属性去可选的获得一个smtpserver属性。

*# content of test\_anothersmtp.py*

smtpserver = "mail.python.org" *# will be read by smtp fixture*

**def** test\_showhelo(smtp):

**assert** 0, smtp.helo()

### 参数化fixtures

### 模块化：从一个fixture函数使用其他fixtures

参数化有助于编写详尽的功能测试组件，本身可以以多种方式配置。

*# content of test\_appsetup.py*

**import** **pytest**

**class** **App**:

**def** \_\_init\_\_(self, smtp):

self.smtp = smtp

**@pytest.fixture**(scope="module")**def** app(smtp):

**return** App(smtp)

**def** test\_smtp\_exists(app):

**assert** app.smtp

### 自动分组测试通过fixture实例

*# content of test\_module.py***import** **pytest**

**@pytest.fixture**(scope="module", params=["mod1", "mod2"])**def** modarg(request):

param = request.param

**print** (" SETUP modarg *%s*" % param)

**def** fin():

**print** (" TEARDOWN modarg *%s*" % param)

request.addfinalizer(fin)

**return** param

**@pytest.fixture**(scope="function", params=[1,2])**def** otherarg(request):

param = request.param

**print** (" SETUP otherarg *%s*" % param)

**def** fin():

**print** (" TEARDOWN otherarg *%s*" % param)

request.addfinalizer(fin)

**return** param

**def** test\_0(otherarg):

**print** (" RUN test0 with otherarg *%s*" % otherarg)**def** test\_1(modarg):

**print** (" RUN test1 with modarg *%s*" % modarg)**def** test\_2(otherarg, modarg):

**print** (" RUN test2 with otherarg *%s* and modarg *%s*" % (otherarg, modarg))