


TOPICS Testing

## 在Visual Studio Code中进行Python测试

 (<https://github.com/Microsoft/vscode-docs/blob/master/docs/python/testing.md>)

在Python扩展 (<https://marketplace.visualstudio.com/items?itemName=ms-python.python>)支持与测试Python的内置的单元测试 (<https://docs.python.org/3/library/unittest.html>) 框架以及pytest (<https://docs.pytest.org/en/latest/>)。尽管框架本身处于维护模式，但也支持鼻子 (<https://nose.readthedocs.io/en/latest/>)。

之后使一个测试框架，使用Python的：**发现测试**命令扫描测试的项目根据当前选定的测试框架的发现模式。一旦发现，Visual Studio Code提供了多种运行测试和调试测试的方法。VS Code在“Python测试日志”面板中显示测试输出，包括未安装测试框架时引起的错误。使用pytest，失败的测试也会出现在“问题”面板中。

### 单元测试的一些背景

(如果您已经熟悉单元测试，则可以跳到演练。)

甲单元是特定的代码进行测试，如函数或类。然后，单元测试就是其他代码片段，这些代码专门使用全部不同的输入（包括边界和边缘情况）来练习代码单元。

例如，假设您具有验证用户以Web形式输入的帐号格式的功能：

```
def validate_account_number_format(account_string):
    # Return false if invalid, true if valid
    # ...
```

单元测试仅与单元的接口（参数和返回值）有关，而与其实现无关（这就是为什么函数体中没有显示代码的原因；通常您会使用其他经过良好测试的库来帮助实现功能）。在此示例中，该函数接受任何字符串，如果该字符串包含格式正确的帐号，则返回true，否则返回false。

为了彻底测试此功能，您想对所有可能的输入都扔给它：有效的字符串，错误键入的字符串（一个或两个字符分开，或包含无效字符），太短或太长的字符串，空白字符串，空参数，包含控制字符的字符串（非文本代码），包含HTML的字符串，包含注入攻击的字符串（例如SQL命令或JavaScript代码）等等。如果经过验证的字符串以后在数据库查询中使用或在应用程序的UI中显示，则测试注入攻击等安全情况就显得尤为重要。

然后，对于每个输入，您定义函数的期望返回值（或多个值）。再次在此示例中，该函数应仅对格式正确的字符串返回true。（数字本身是否是真实帐户是另一回事，将通过数据库查询在其他地方处理。）

掌握了所有参数和期望的返回值之后，您现在可以自己编写测试，这些代码是使用特定输入调用函数的代码，然后将实际的返回值与期望的返回值进行比较（此比较称为断言）：

```
# Import the code to be tested
import validator

# Import the test framework (this is a hypothetical module)
import test_framework

# This is a generalized example, not specific to a test framework
class Test_TestAccountValidator(test_framework.TestBaseClass):
    def test_validator_valid_string():
        # The exact assertion call depends on the framework as well
        assert(validate_account_number_format("1234567890"), true)

    # ...

    def test_validator_blank_string():
        # The exact assertion call depends on the framework as well
        assert(validate_account_number_format(""), false)

    # ...

    def test_validator_sql_injection():
        # The exact assertion call depends on the framework as well
        assert(validate_account_number_format("drop database master"), false)

    # ... tests for all other cases
```

代码的确切结构取决于您使用的测试框架，本文稍后提供了具体示例。无论如何，如您所见，每种测试都非常简单：使用参数调用函数并声明期望的返回值。

所有测试的合并结果就是您的测试报告，该报告可以告诉您该功能（单元）在所有测试用例中是否表现出预期的效果。也就是说，当一个单元通过所有测试时，您可以确信其功能正常。（*测试驱动开发*的实践是首先实际编写测试，然后编写代码以通过越来越多的测试，直到所有这些都通过为止。）

因为单元测试是小的，独立的代码段（在单元测试中，您避免了外部依赖关系，而是使用模拟数据或其他模拟输入），所以它们运行起来既快捷又便宜。此特性意味着您可以尽早且经常运行单元测试。开发人员通常甚至在将代码提交到存储库之前就运行单元测试。登机检入系统还可以在合并提交之前运行单元测试。每次构建后，许多连续集成系统也会运行单元测试。尽早运行单元测试，通常意味着您可以快速掌握~~回归~~，这是先前通过其所有单元测试的代码的行为的意外更改。因为可以轻松地将测试失败追溯到特定的代码更改，所以很容易找到并纠正失败的原因，这无疑比在此过程的后面发现问题要好得多！

有关单元测试的一般背景，看到单元测试 ([https://wikipedia.org/wiki/Unit\\_testing](https://wikipedia.org/wiki/Unit_testing)) 维基百科。有关各种有用的单元测试示例，请参见<https://github.com/gwtw/py-sorting> (<https://github.com/gwtw/py-sorting>)，该库包含用于不同排序算法的测试。

### 测试演练示例

Python测试是Python类，它们与要测试的代码位于不同的文件中。每个测试框架都指定测试和测试文件的结构和命名。一旦编写了测试并启用了测试框架，VS Code就会找到这些测试并为您提供各种命令来运行和调试它们。

对于本节，创建一个文件夹并在VS Code中打开它。然后 `inc_dec.py` 使用以下代码创建一个文件进行测试：

```
def increment(x):
    return x + 1

def decrement(x):
    return x - 1
```

使用此代码，您可以体验VS代码中的测试，如以下各节所述。

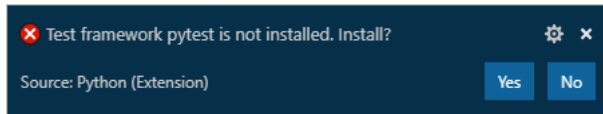
## 启用测试框架

默认情况下，禁用Python测试。要启用测试，请使用“命令面板”上的Python：**配置测试**命令。此命令提示您选择测试框架，包含测试的文件夹以及用于标识测试文件的模式。

您也可以手动配置测试通过设置一个且只有一个以下设置为true：，`python.testing.unittestEnabled`，`python.testing.pytestEnabled` 和 `python.testing.nosetestsEnabled`。每个框架还具有特定的配置设置，如其文件夹和模式的“测试配置设置”中所述。

一次只能启用一个测试框架非常重要。因此，当您启用一个框架时，还请确保禁用其他框架。在Python的：**配置测试**命令自动执行此操作。

启用测试框架时，如果当前激活的环境中不存在该框架软件包，则VS Code会提示您安装该框架软件包：



## 创建测试

每个测试框架都有自己的约定，用于命名测试文件和在其中构造测试，如以下各节所述。每个案例都包含两种测试方法，其中一种出于演示目的而故意设置为失败。

由于Nose处于维护模式，不建议用于新项目，因此以下部分仅显示unittest和pytest示例。（Nose2的后继者Nose2只是带有插件的单元测试，因此它遵循此处显示的单元测试模式。）

### 单元测试中的测试

创建一个名为的文件 `test_unittest.py`，其中包含带有两个测试方法的测试类：

```
import inc_dec    # The code to test
import unittest   # The test framework

class Test_TestIncrementDecrement(unittest.TestCase):
    def test_increment(self):
        self.assertEqual(inc_dec.increment(3), 4)

    def test_decrement(self):
        self.assertEqual(inc_dec.decrement(3), 4)

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

### pytest中的测试

创建一个名为的文件 `test_pytest.py`，其中包含两种测试方法：

```
import inc_dec    # The code to test

def test_increment():
    assert inc_dec.increment(3) == 4

def test_decrement():
    assert inc_dec.decrement(3) == 4
```

## 测试发现

VS Code使用当前启用的测试框架来发现测试。您可以随时使用Python：**发现测试**命令来触发测试发现。

`python.testing.autoTestDiscoverOnSaveEnabled` true 默认情况下设置为，这意味着只要保存测试文件，就会自动执行测试发现。要禁用此功能，请将值设置为 false。

测试发现将发现模式应用于当前框架（可以使用测试配置设置进行自定义）。默认行为如下：

- `python.testing.unittestArgs`：查找 `.py` 顶级项目文件夹中名称中带有“test”的任何Python（）文件。所有测试文件必须是可导入的模块或软件包。您可以使用 `-p` 配置设置来自定义文件匹配模式，并使用该 `-t` 设置来自定义文件夹。
- `python.testing.pytestArgs`：查找 `.py` 位于当前文件夹及其所有子文件夹内任何位置的名称以“test”开头或以“\_test”结尾的任何Python（）文件。

**提示：**有时找不到放置在子文件夹中的测试，因为无法导入此类测试文件。要使其可导入，请 `__init__.py` 在该文件夹中创建一个空文件。

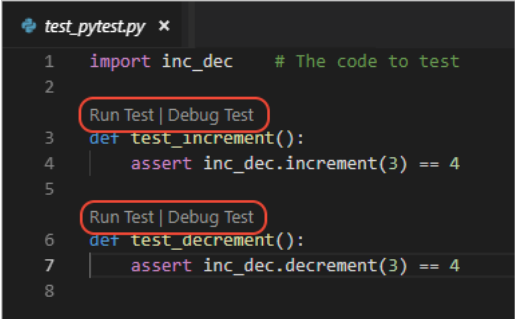
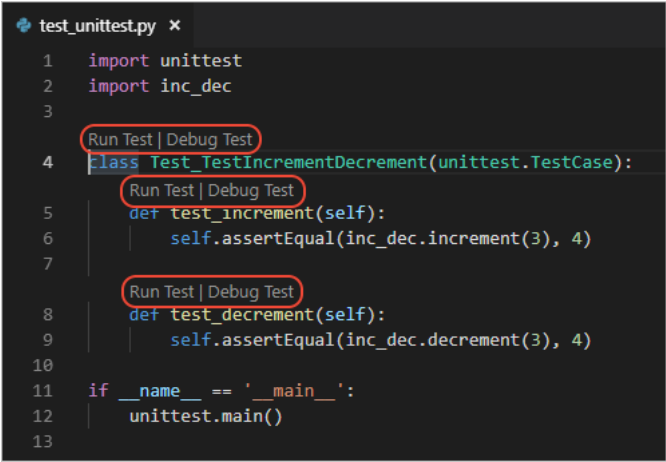
如果发现成功，状态栏将显示“运行测试”：



如果发现失败（例如，未安装测试框架），则会在状态栏上看到一条通知。选择通知可提供更多信息：

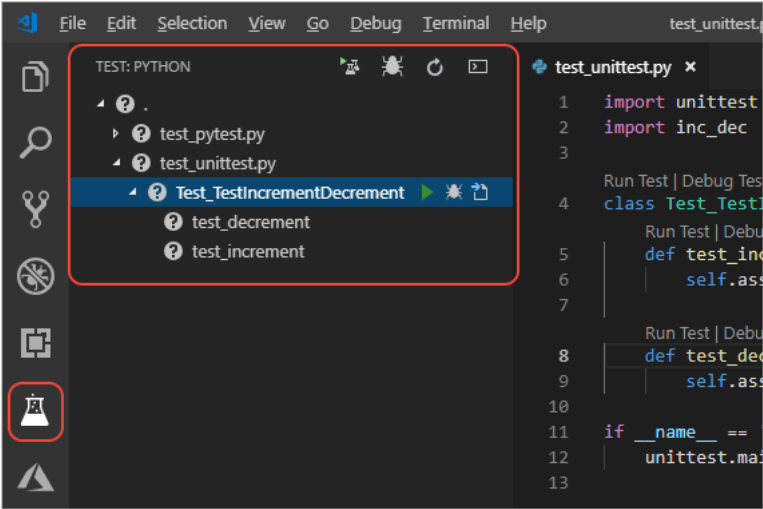


VS Code识别测试后，它会提供几种运行这些测试的方法，如“运行测试”中所述。最明显的方法是直接在编辑器中显示的CodeLens装饰，使您可以轻松地运行单个测试方法或使用unittest运行测试类：



**注意：**目前，Python扩展没有提供打开或关闭装饰的设置。要提出不同的行为，请在vscode-python信息库中提交问题 (<https://github.com/Microsoft/vscode-python/issues>)。

对于Python，测试发现还会通过VS Code活动栏上的图标激活“测试资源管理器”。该测试资源管理器可以帮助您显现，导航和运行测试：



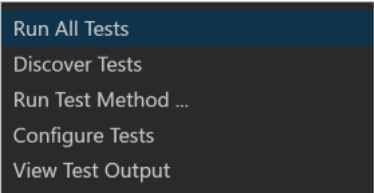
运行测试

您可以使用以下任一操作来运行测试：

- 打开测试文件后，选择出现在测试方法或类上方的“运行测试代码镜头”装饰，如上一节所示。此命令仅运行一个方法或类中的那些测试。
- 选择状态栏上的“运行测试”（可以根据结果更改外观），



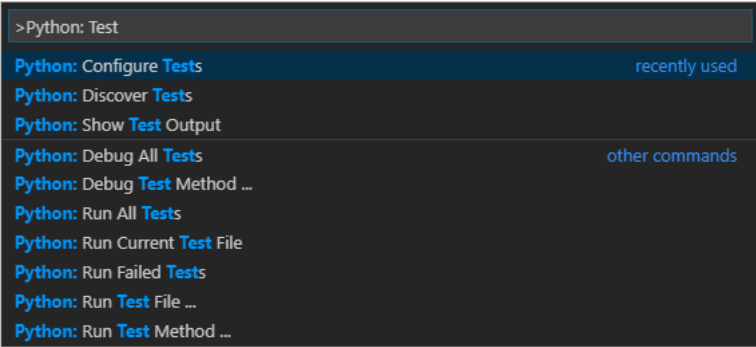
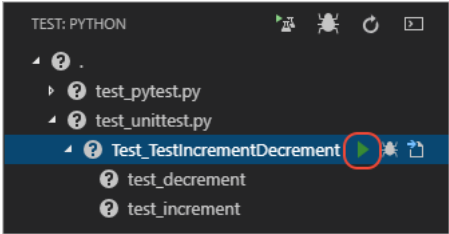
然后选择“运行所有测试”或“发现测试”之类的命令之一：



- 在**测试资源管理器**中：
  - 要运行所有发现的测试，请选择“**测试资源管理器**”顶部的“播放”按钮：

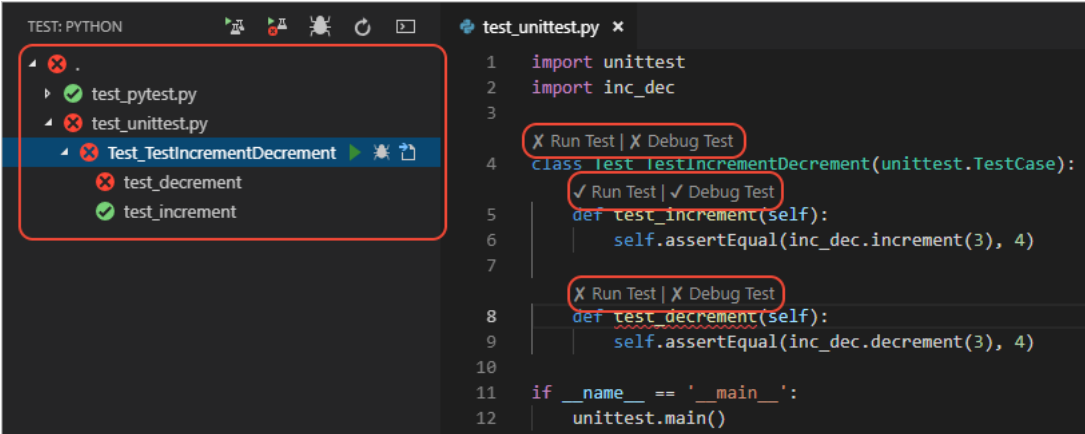


- 要运行一组特定的测试或一个测试，请选择文件，类或测试，然后选择该项目右侧的播放按钮：
- 在资源管理器中右键单击一个文件，然后选择Run All Tests，它将在该文件中运行测试。
  - 在**命令面板**中，选择任何运行测试命令：

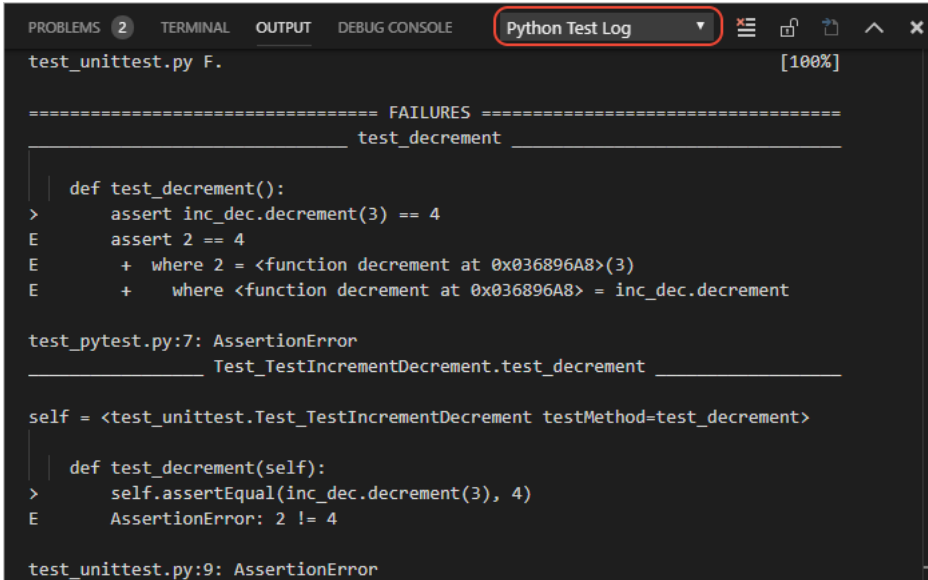


命令	描述
调试所有测试	请参阅调试测试。
调试测试方法	请参阅调试测试。
运行所有测试	在工作区及其子文件夹中搜索并运行所有测试。
运行当前测试文件	在编辑器中当前查看的文件中运行测试。
运行失败的测试	重新运行在之前的测试运行中失败的所有测试。如果尚未运行任何测试，则运行所有测试。
运行测试文件	提示输入特定的测试文件名，然后在该文件中运行测试。
运行测试方法	提示要运行的测试名称，并自动完成测试名称。
显示测试输出	打开“Python测试日志”面板，其中包含有关通过和失败测试以及错误和跳过的测试的信息。

测试运行后，VS Code在编辑器和Test Explorer中直接使用CodeLens装饰显示结果。将显示单个测试以及包含这些测试的所有类和文件的结果。测试失败时，还会在编辑器中用红色下划线装饰。



VS Code还在Python测试日志输出面板中显示测试结果（使用“查看”>“输出”菜单命令显示“输出”面板，然后从右侧的下拉菜单中选择“Python测试日志”）：



```
test_unittest.py F. [100%]

===== FAILURES =====
_____ test_decrement _____

    def test_decrement():
>     assert inc_dec.decrement(3) == 4
E       assert 2 == 4
E       + where 2 = <function decrement at 0x036896A8>(3)
E       + where <function decrement at 0x036896A8> = inc_dec.decrement

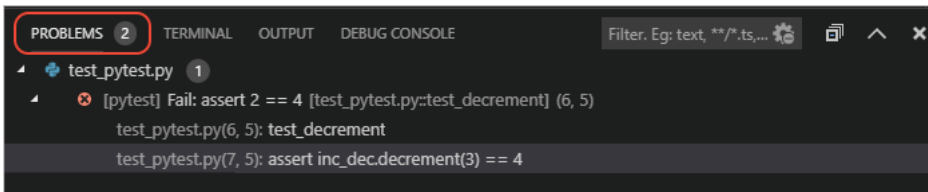
test_pytest.py:7: AssertionError
_____ Test_TestIncrementDecrement.test_decrement _____

self = <test_unittest.Test_TestIncrementDecrement testMethod=test_decrement>

    def test_decrement(self):
>     self.assertEqual(inc_dec.decrement(3), 4)
E       AssertionError: 2 != 4

test_unittest.py:9: AssertionError
```

使用pytest，失败的测试也将出现在“问题”面板中，您可以在其中双击问题直接导航到测试：



```
PROBLEMS 2 | TERMINAL | OUTPUT | DEBUG CONSOLE | Filter: Eg: text, **/*.ts,...

test_pytest.py 1
  [pytest] Fail: assert 2 == 4 [test_pytest.py:test_decrement] (6, 5)
    test_pytest.py(6, 5): test_decrement
    test_pytest.py(7, 5): assert inc_dec.decrement(3) == 4
```

## 并行运行测试

该 pytest-xdist 软件包提供了与pytest并行运行测试的支持。要启用并行测试：

1. 打开集成终端并安装 pytest-xdist 软件包。有关更多详细信息，请参阅项目的文档页面 (<https://pypi.org/project/pytest-xdist/>)。

### 对于窗户

```
py -3 -m pip install pytest-xdist
```

### 对于macOS / Linux

```
python3 -m pip install pytest-xdist
```

2. 接下来，pytest.ini 在项目目录中创建一个名为的文件，并在下面添加内容，并指定要使用的CPU数量。例如，要将其设置为4个CPU：

```
[pytest]
addopts=-n4
```

3. 运行您的测试，现在将并行运行。

## 调试测试

因为测试本身是源代码，所以它们就像测试的生产代码一样容易产生代码缺陷。因此，您有时可能需要逐步调试并分析调试器中的测试。

例如，test\_decrement 前面给出的功能失败，因为断言本身是错误的。以下步骤演示了如何分析测试：

1. 在 test\_decrement 函数的第一行上设置一个断点。
2. 在“测试资源管理器”中选择该功能上方的“调试测试”装饰或该测试的“错误”图标。VS Code启动调试器，并在断点处暂停。
3. 在“调试控制台”面板中，输入 inc\_dec.decrement(3) 以查看实际结果是2，而在测试中指定的预期结果是不正确的值4。
4. 停止调试器并更正错误的代码：

```
# unittest
self.assertEqual(inc_dec.decrement(3), 2)

# pytest
assert inc_dec.decrement(3) == 2
```

5. 保存该文件，然后再次运行测试以确认它们已通过，并看到CodeLens装饰也指示通过状态。

**注意：**运行或调试测试不会自动保存测试文件。在运行测试之前，请务必确保将更改保存到测试中，否则结果可能会使您感到困惑，因为它们仍然反映文件的先前版本！

“ Python: **调试所有测试**”和“ Python: **Debug测试方法**”命令（在“命令面板”和“状态栏”菜单上）分别为所有测试和单个测试方法启动调试器。您还可以使用“ **测试资源管理器** ”中的“错误”图标为选定范围内的所有测试以及所有发现的测试启动调试器。

调试器与其他Python代码（包括断点，变量检查等）的测试工作相同。要自定义调试测试的设置，可以 `"request": "test"` 在 `.vscode` 工作区的文件夹的`launch.json`文件中指定。当您运行Python: **调试所有测试**和Python: **调试测试方法**命令时，将使用此配置。

例如， `launch.json` 文件中下面的配置将禁用 `justMyCode` 调试测试的设置：

```
{
  "name": "Debug Tests",
  "type": "python",
  "request": "test",
  "console": "integratedTerminal",
  "justMyCode": false
}
```

如果您有多个配置条目 `"request": "test"` ，则将使用第一个定义，因为我们当前不支持此请求类型的多个定义。

有关调试的更多信息，请参见Python调试配置 (`/docs/python/debugging`)和常规VS代码调试 (`/docs/editor/debugging`)文章。

## 测试配置设置

使用Python进行测试的行为由常规设置和特定于您启用的框架的设置共同驱动。

### 常规设置

设置 ( <code>python.testing.</code> )	默认	描述
<code>autoTestDiscoverOnSaveEnabled</code>	<code>true</code>	指定保存测试文件时是启用还是禁用自动运行测试发现。
电脑	空值	指定用于测试的可选工作目录。
调试端口	<code>3000</code>	用于调试单元测试测试的端口号。
提示配置	<code>true</code>	指定如果发现潜在的测试，VS Code是否提示配置测试框架。

### 单元测试配置设置

设置 ( <code>python.testing.</code> )	默认	描述
<code>unittestEnabled</code>	<code>false</code>	指定是否启用unittest作为测试框架。所有其他框架都应禁用。
<code>unittestArgs</code>	<code>["-v", "-s", ".", "-p", "*test*.py"]</code>	传递给unittest的参数，其中用空格分隔的每个元素都是列表中的单独项目。有关默认值的说明，请参见下文。

unittest的默认参数如下：

- `-v` 设置默认的详细程度。删除此参数以获得更简单的输出。
- `-s` . 指定用于发现测试的起始目录。如果在“ `test` ”文件夹中有测试，请将参数更改为 `-s test` （ `"-s", "test"` 在 `arguments` 数组中表示）。
- `-p *test*.py` 是用于查找测试的发现模式。在这种情况下，它是任何 `.py` 包含单词“ `test` ”的文件。如果您以不同的方式命名测试文件，例如在每个文件名后附加“ `_test` ”，则使用类似于 `*_test.py` 数组适当参数的模式。

要在第一次失败时停止测试运行，请将`fail fast`选项添加 `"-f"` 到`arguments`数组。

有关可用选项的完整集合，请参见unittest命令行界面 (<https://docs.python.org/3/library/unittest.html#command-line-interface>)。

### pytest配置设置

设置 ( <code>python.testing.</code> )	默认	描述
<code>pytestEnabled</code>	<code>false</code>	指定是否启用pytest作为测试框架。所有其他框架都应禁用。
<code>pytestPath</code>	<code>"pytest"</code>	pytest的路径。如果pytest位于当前环境之外，请使用完整路径。
<code>pytestArgs</code>	<code>[]</code>	传递给pytest的参数，其中用空格分隔的每个元素都是列表中的单独项目。请参阅pytest命令行选项 ( <a href="https://docs.pytest.org/en/latest/customize.html#command-line-options-and-configuration-file-settings">https://docs.pytest.org/en/latest/customize.html#command-line-options-and-configuration-file-settings</a> )。

您还可以使用pytest Configuration中 (<https://docs.pytest.org/en/latest/customize.html>) `pytest.ini` 所述的文件来配置pytest (<https://docs.pytest.org/en/latest/customize.html>)。

**注意** 如果安装了 `pytest-cov coverage` 模块，则VS Code在调试时不会在断点处停止，因为 `pytest-cov` 使用相同的技术来访问正在运行的源代码。为防止此行为，请 `--no-cov` 在 `pytestArgs` 调试测试时将其包括在内。（有关更多信息，请参阅 `pytest-cov` 文档中的 `Debuggers` 和 `PyCharm` (<https://pytest-cov.readthedocs.io/en/latest/debuggers.html>)。）

鼻子配置设置 #

设置 (python.testing.)	默认	描述
鼻子测试启用	false	指定是否启用“鼻子”作为测试框架。所有其他框架都应禁用。
鼻子测试路径	"nosetests"	鼻子的路径。如果鼻子位于当前环境之外，请使用完整路径。
鼻子测试	[]	传递给Nose的参数，其中用空格分隔的每个元素都是列表中的单独项目。请参阅鼻子用法选项 (https://nose.readthedocs.io/en/latest/usage.html#options)。

您还可以按照鼻子配置中 (https://nose.readthedocs.io/en/latest/usage.html#configuration)所述用 .noserc 或 nose.cfg 文件配置鼻子。  
(https://nose.readthedocs.io/en/latest/usage.html#configuration)

也可以看看

- Python环境 (/docs/python/environments) -控制使用哪个Python解释器进行编辑和调试。
- 设置参考 (/docs/python/settings-reference) -在VS Code中探索与Python相关的所有设置。

该文档对您有帮助吗？

是

没有

2019/06/28

在这篇文章中

- 单元测试的一些背景
- 测试演练示例
- 启用测试框架
- 创建测试
- 测试发现
- 运行测试
- 并行运行测试
- 调试测试

{ 测试配置设置

也可以看看

- 鸣叫此链接

(https://twitter.com/intent/tweet?original\_referer=https://code.visualstudio.com/docs/python/testing&ref\_src=twsrc%5Etfw&text=Testing%20Python%20in%20Visual%20Studio%20Code&tw\_p=tweet)
- 订阅

(/feed.xml)
- 问问题

(https://stackoverflow.com/questions/tagged/vscode)
- 按照@code

(https://go.microsoft.com/fwlink/?LinkID=533687)
- 请求功能

(https://go.microsoft.com/fwlink/?LinkID=533482)
- 报告问题

(https://www.github.com/Microsoft/vscode/issues)
- 看视频

(https://www.youtube.com/channel/UCs5Y5\_7XK8HLDX0SLNwkd3w)

你好，西雅图。 按照@code (https://go.microsoft.com/fwlink/?LinkID=533687) 

星