

第一章 unittest 单元测试框架

unittest 是 python 自带的测试框架，主要适用于单元测试，可以对多个测试用例进行管理和封装，并通过执行输出测试结果，接下来就开始介绍这个框架。

1.1. 认识 unittest

unittest 模块是 python 标准库中的模块，其模块提供了许多类和方法处理各种测试工作。

- 一个测试用例基类 (TestCase)

unittest 提供了一个测试用例基类 TestCase，我们编写的测试用例都必须继承自 TestCase。

TestCase 提供 assertxxx 方法，用于将执行的结果与预期的结果进行检验，匹配或不匹配，都会被系统记录下来最后形成报告。

TestCase 还包含有 setUp, tearDown 方法用于在执行测试用例之前及之后进行指定的操作。

- TestSuite

将一组相关的测试用例放在一起，形成一个有意义的 TestSuite，TestSuite 负责保存、管理（添加/删除）多个单元测试。

- 一个测试运行器 (TestRunner)

测试运行器负责按约定的（或用户指定的）配置装载测试，然后执行测试，生成报告。

- TestReport (测试报告)

测试报告用来展示所有执行用例的成功或者失败状态的汇总，执行失败的测试步骤的预期结果与实际结果，还有整体运行状况和运行时间的汇总。

以上这些构筑了整个 unittest 的测试框架结构，框架如图 9-1 所示。

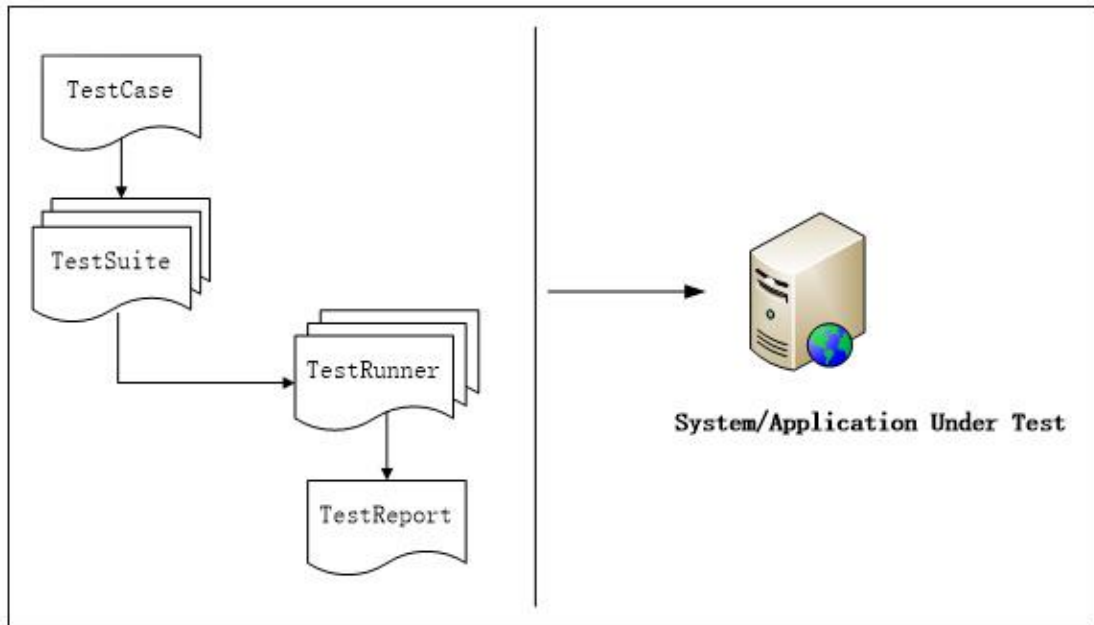


图 9-1 unittest 测试框架

1.2. Python 安装

本书是基于 Windows 平台开发 Python 程序的。本节接下来分步骤给大家演示如何在 Windows 平台下安装 Python 开发环境，具体如下。

(1) 访问 <http://www.python.org/download>，选择 Windows 平台下的安装包，如图 9-2 所示。

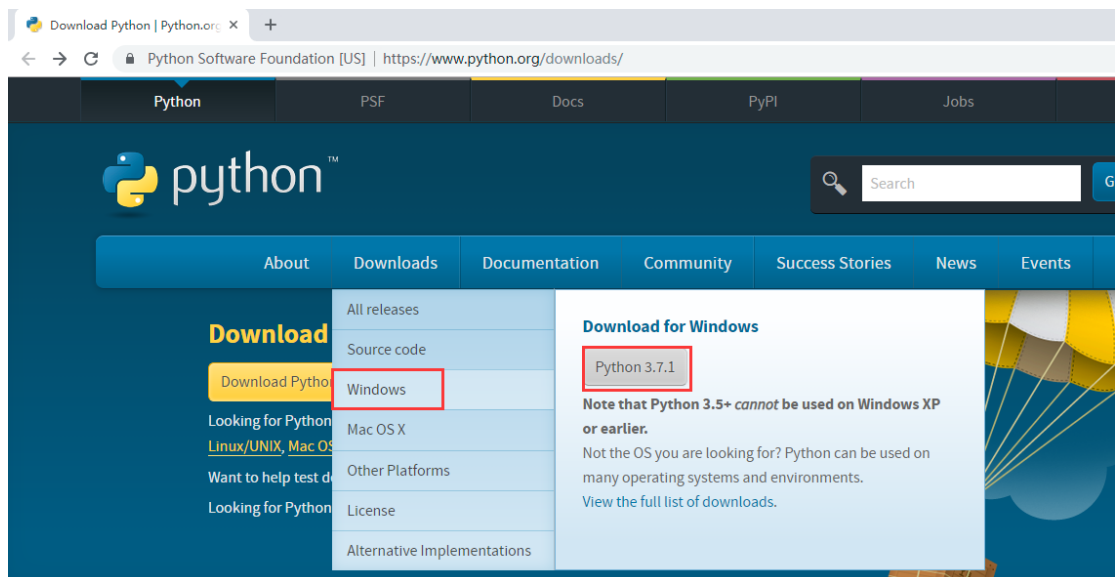


图 9-2 选择 Windows 平台的安装包

(2) 单击图 1-1 中的 Python3.7.1 进行下载，下载后的文件名为“python-3.7.1.exe”。双击该文件，进入安装 Python 的界面，如图 9-3 所示。

在图 9-3 中，提示有两种安装方式。第一种是采用默认的安装方式，第二种是自定义安装方式，可以自己选择软件的安装路径。这两种安装方式都可以。

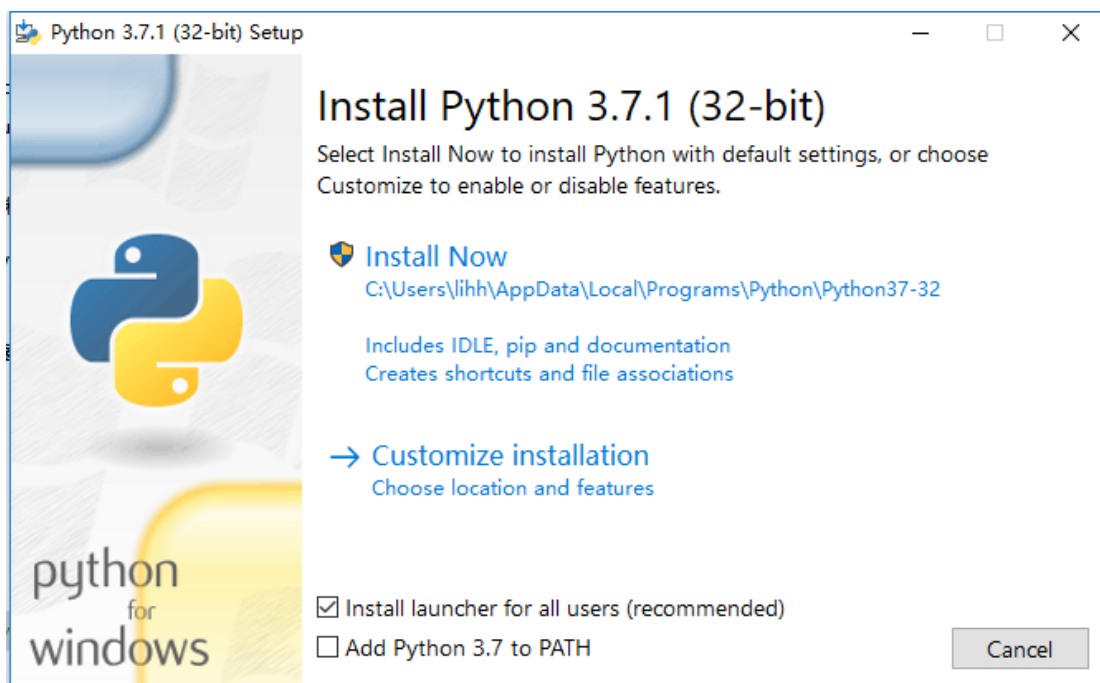


图 9-3 选择安装方式

(3) 这里我们选择第一种安装方式，安装界面如图 9-4 所示。

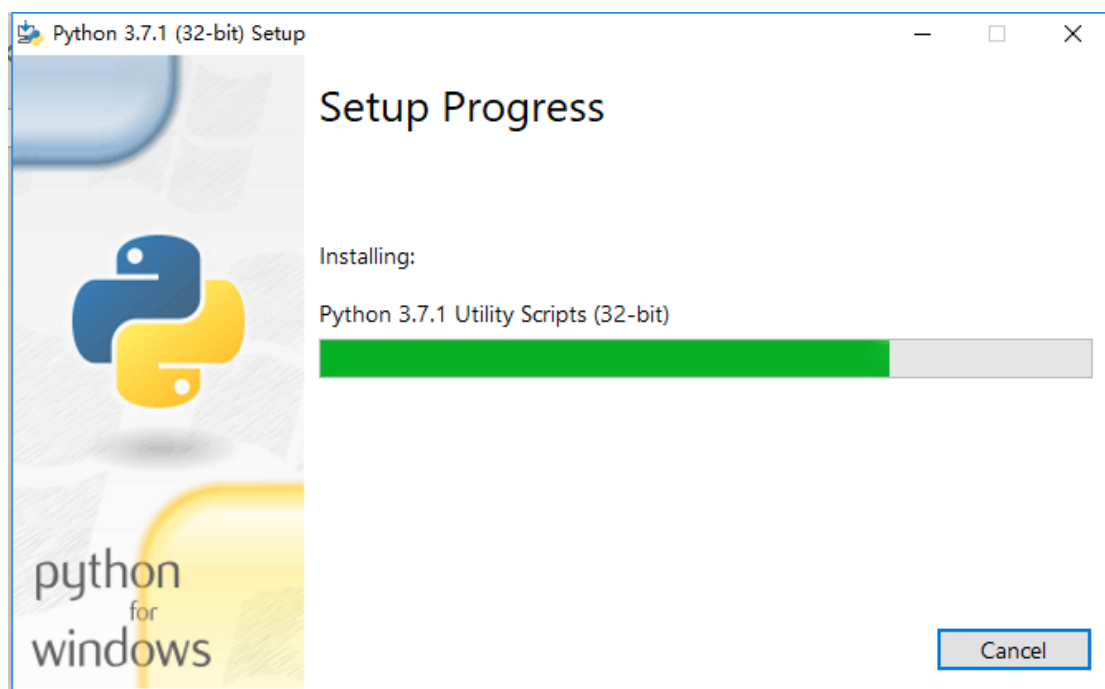


图 9-4 安装界面

(4) Python 的安装进度非常快，安装成功后的界面如图 9-5 所示。

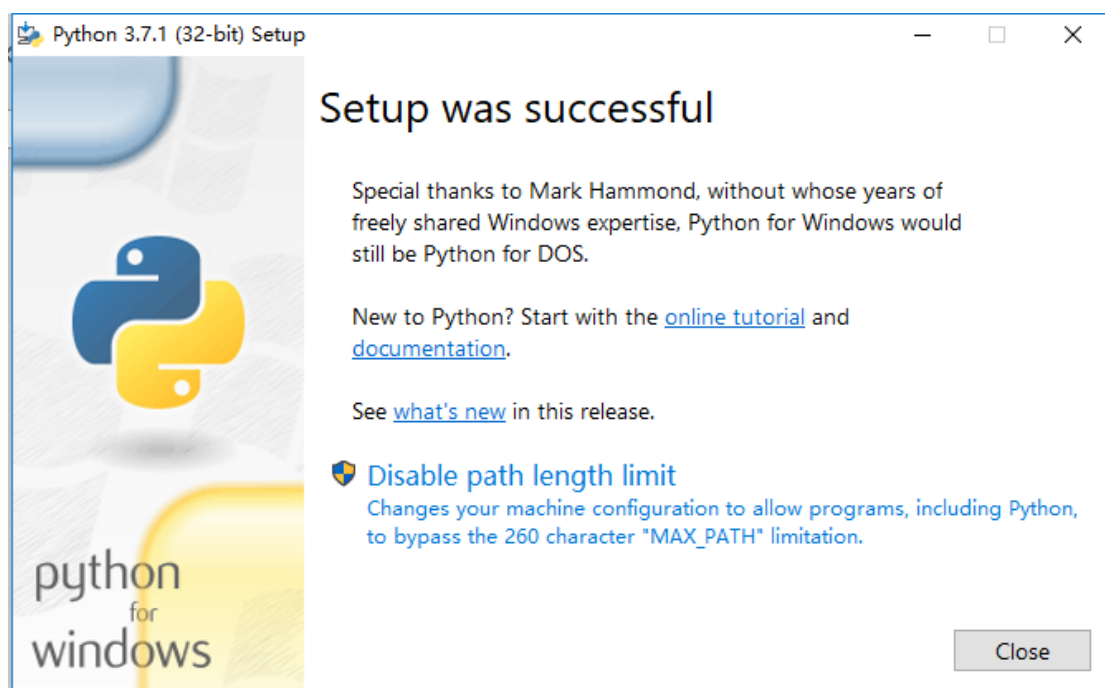


图 9-5 安装成功的界面

这里需要提醒大家一点，在图 9-3 选择安装方式时，最下面有个选项【Add Python 3.7 to PATH】，如果大家勾选了这个选项，那么后续配置环境变量的步

骤可以省略。但是很多时候，大家都会遗漏勾选这个选项。这个时候，就需要我们手动配置环境变量。

(5) 手动添加环境变量。鼠标右击【计算机】→【属性】→【高级系统设置】，弹出图 9-6 所示的【系统属性】对话框。

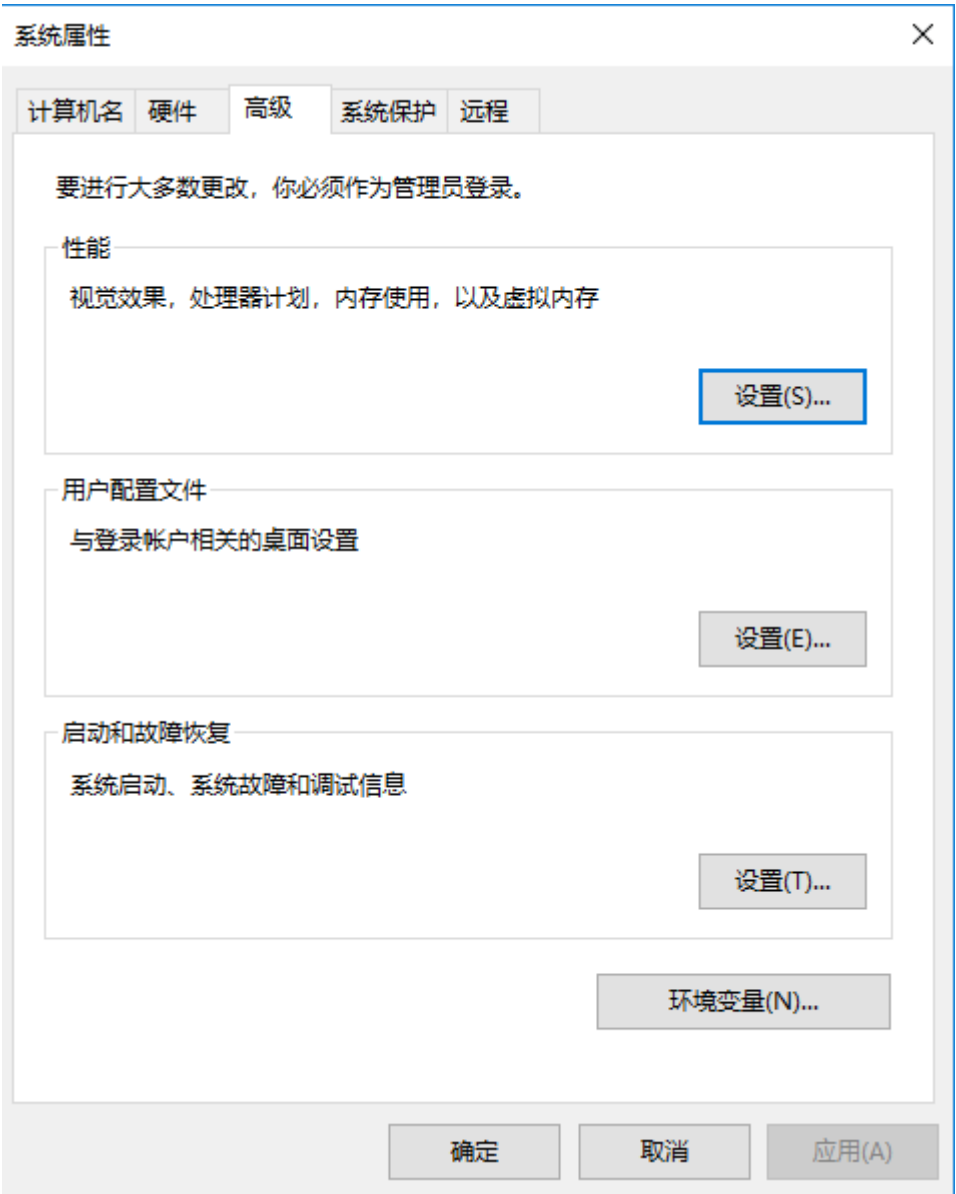


图 9-6 系统属性设置

(6) 单击图 9-6 中的【环境变量】，在弹出的【环境变量】对话框中双击【Path】，如图 9-7 所示。

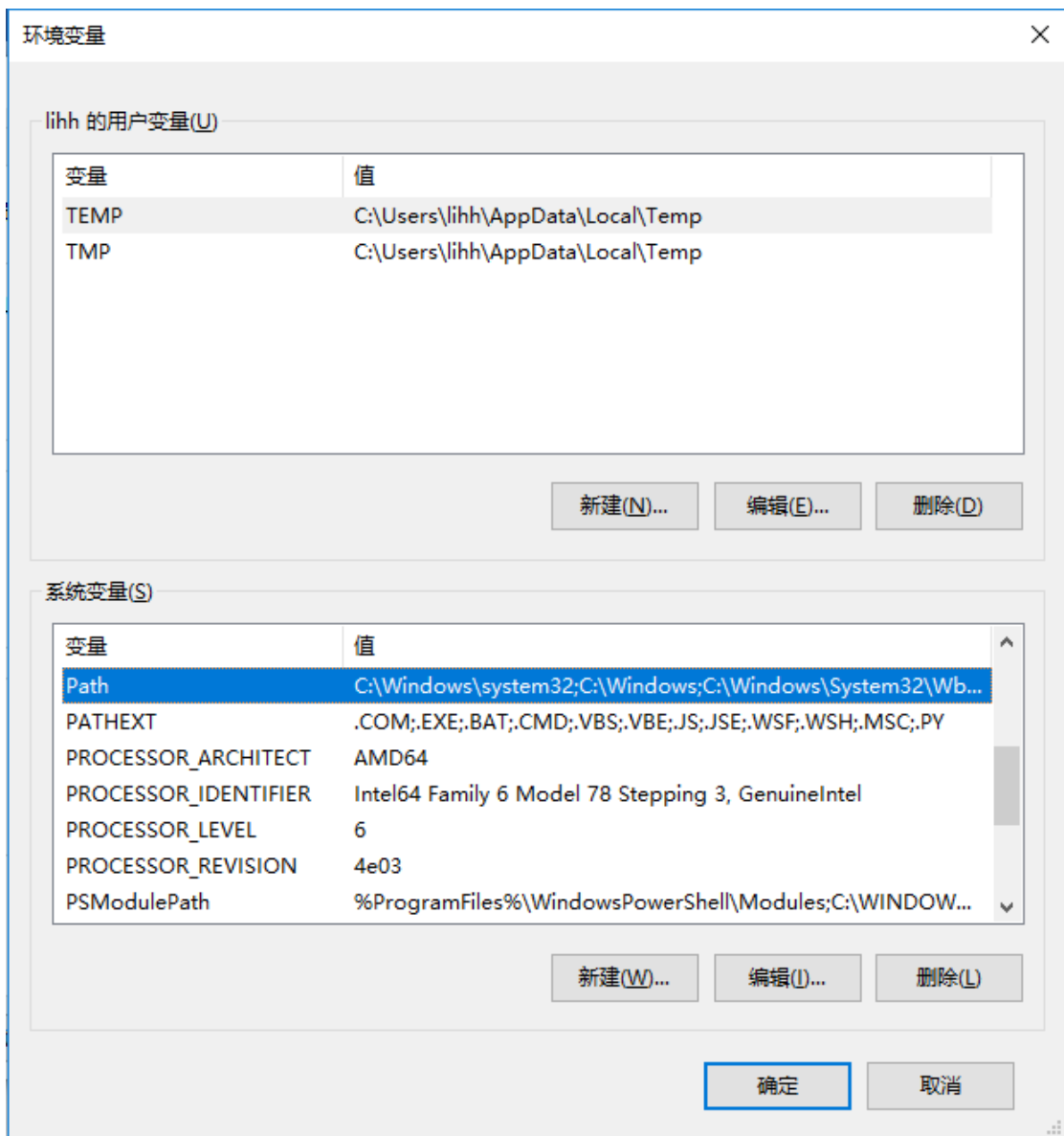


图 9-7 设置环境变量

(7) 在弹出的【编辑环境变量】弹出框中，点击新建，输入 Python 的安装路径，单击【确定】，完成环境变量的配置，如图 9-8 所示。

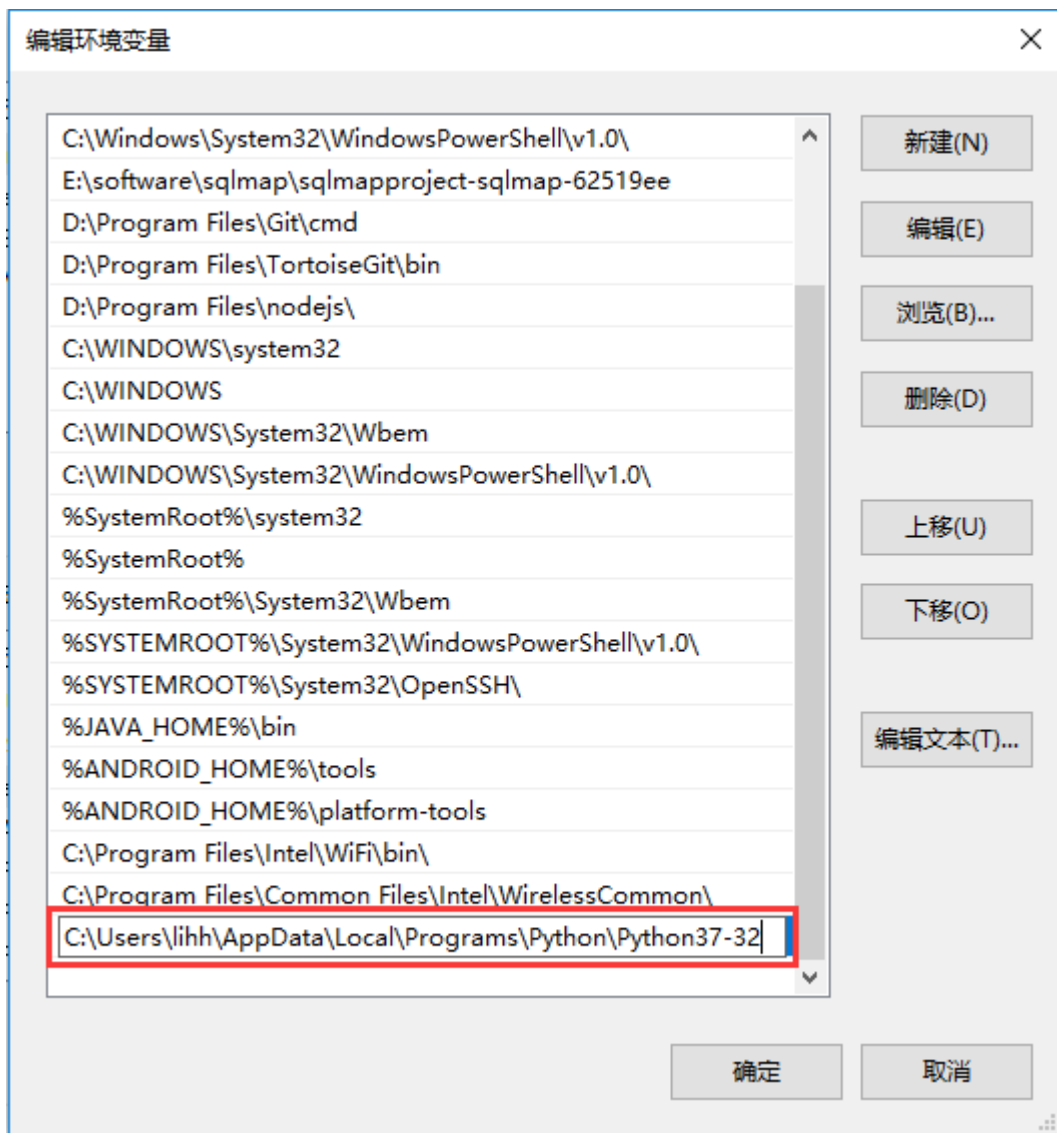


图 9-8 编辑系统环境变量

(7) 此时，在控制台输入“python”，控制台会打印出 Python 的版本信息，如图 9-9 所示。

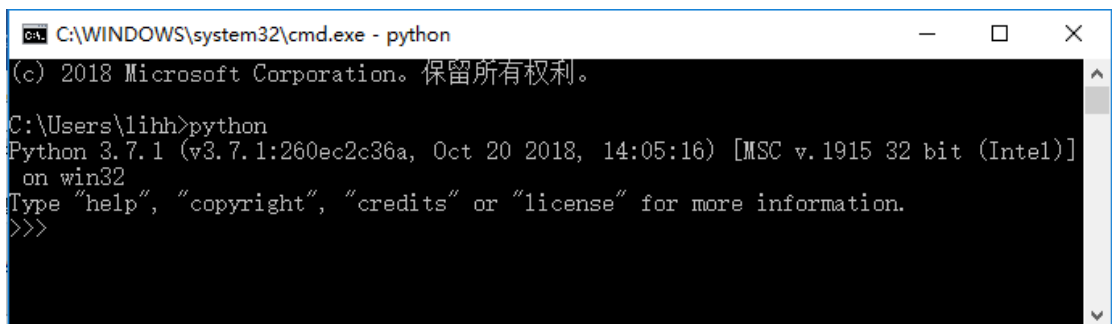


图 9-9 环境变量配置成功后的控制台输出

(8) 配置 pip，我们直接在命令行输入“pip list”，同样会显示‘pip’不是内部命令，也不是可运行的程序，如图 9-10 所示。

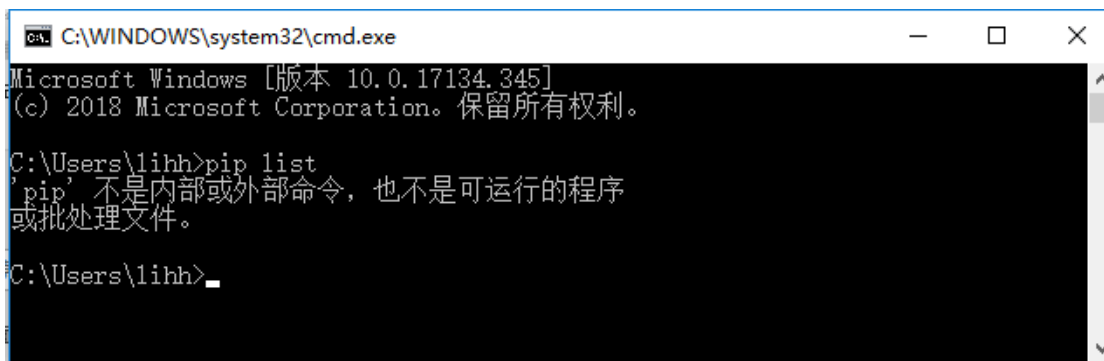


图 9-10 控制台显示 pip 不是内部命令

之所以出现上述问题，是因为我们还没有添加环境变量。按照之前介绍的添加环境变量的方法，我们在 Path 最后面添加 Scripts 文件所在路径，如下所示：

```
C:\Users\lihh\AppData\Local\Programs\Python\Python37-32\Scripts
```

再次打开控制台，输入“pip list”，控制台的输出结果如图 9-11 所示。此时，pip 才被成功安装。



图 9-11 成功安装 pip

这里提醒大家一点，pip 是 Python 包管理工具，该工具提供了对 Python 包的查找、下载、安装、卸载的功能。Python 2.7.9 + 或 Python 3.4+ 以上版本都自带 pip 工具，目前如果我们在 [python.org](https://www.python.org) 下载最新版本的安

装包，则是已经自带了该工具。

1.3. 集成开发环境—PyCharm

PyCharm 是 JetBrains 开发的 Python IDE。PyCharm 具备一般 IDE 的功能，如调试、语法高亮、Project 管理、代码跳转、智能提示、自动完成、单元测试、版本控制等。接下来，本节将针对 PyCharm 的下载安装和使用进行介绍。

1.3.1. PyCharm 的下载安装

访问 PyCharm 的官方网址 <http://www.jetbrains.com/pycharm/download/>，进去 PyCharm 的下载页面，如图 9-12 所示。

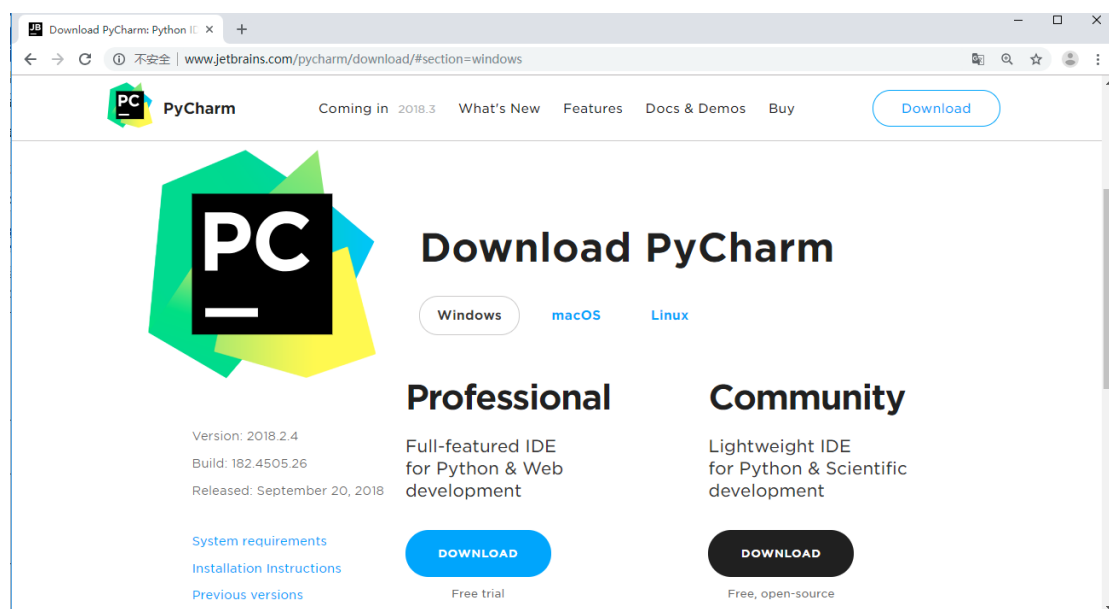


图 9-12 下载 PyCharm 页面

图 9-12 中，我们可以根据不同的平台下载 PyCharm，并且每个平台可以选择下载 Professional 和 Community 两个版本。这里，我们下载 Community 版本。

下载成功后，安装 PyCharm 的过程很简单，只需要运行下载的安装程序，按照安装向导提示一步一步操作即可。这里以 Windows 为例，讲解如何安装 PyCharm，具体步骤如下。

(1) 双击下载好的 exe 安装文件，进入安装 PyCharm 的界面，如图 1-15

所示。



图 1-15 进入安装 PyCharm 界面

(2) 单击图 1-15 中的【Next>】按钮，进入选择安装目录的界面，如图 1-16 所示。

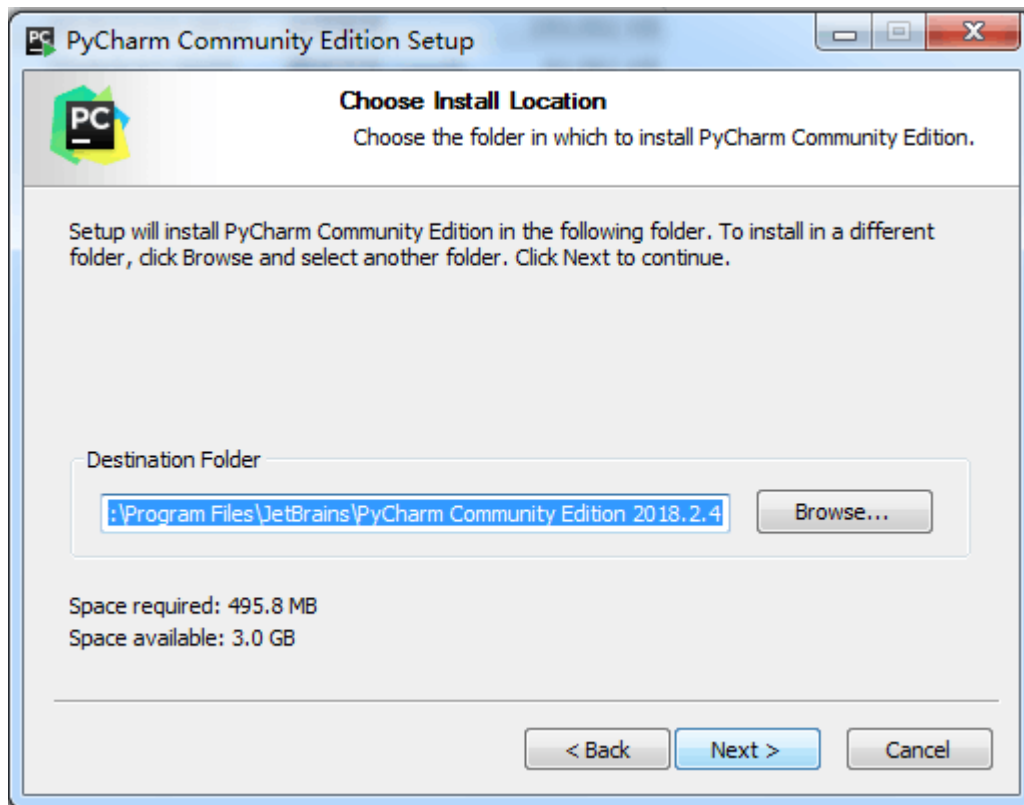


图 1-16 选择 PyCharm 安装的路径

(3) 单击图 1-16 中的【Next>】按钮，进入文件配置的界面，如图 1-17 所示。

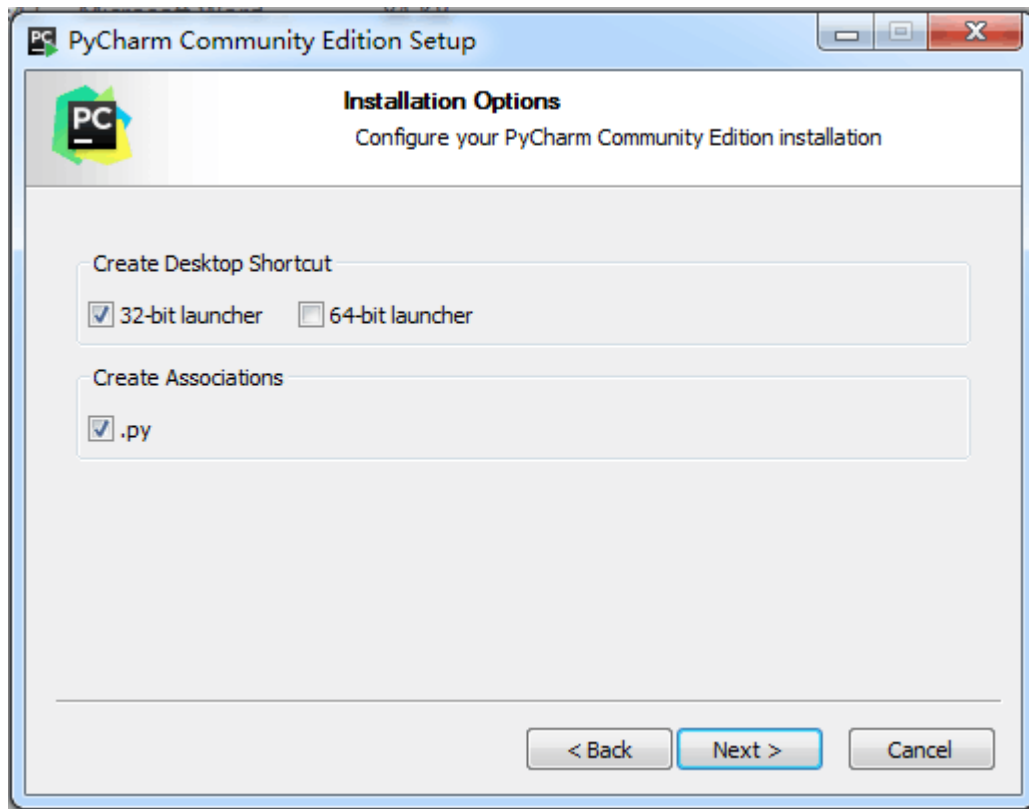


图 1-17 文件配置的相关界面

(4) 单击图 1-17 中的【Next>】按钮，进入选择启动菜单的界面，如图 1-18 所示。

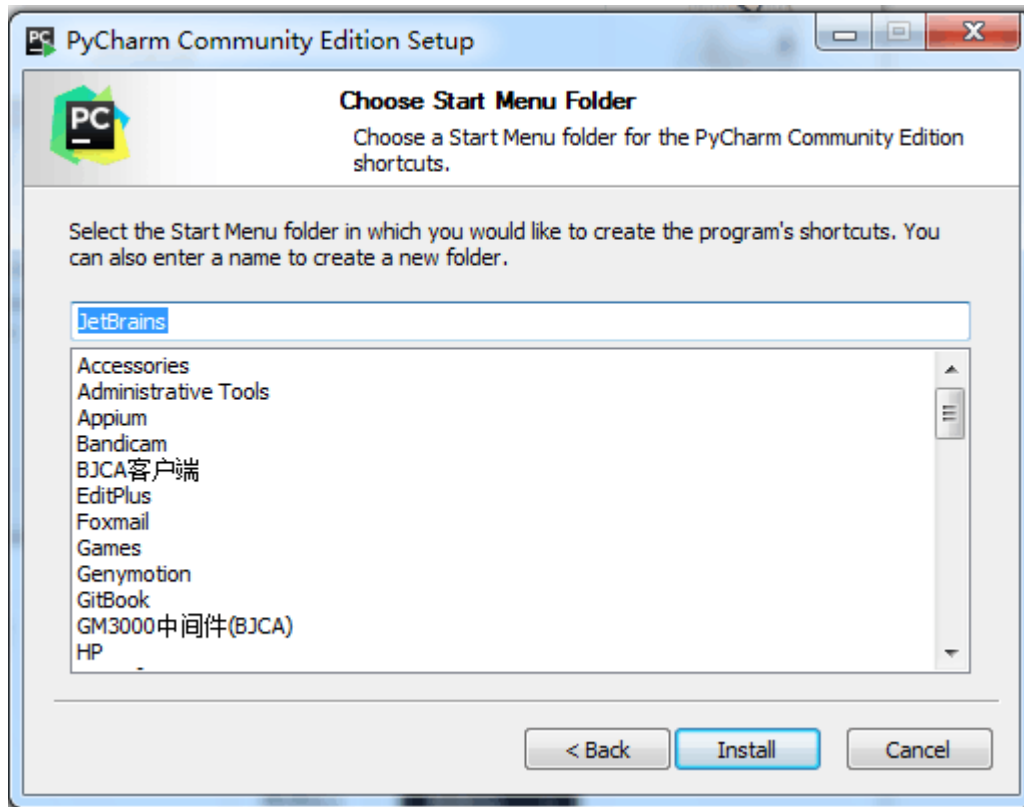


图 1-18 选择启动菜单文件

(5) 单击图 1-18 中的【Install】按钮，开始安装 PyCharm，如图 1-19 所示。

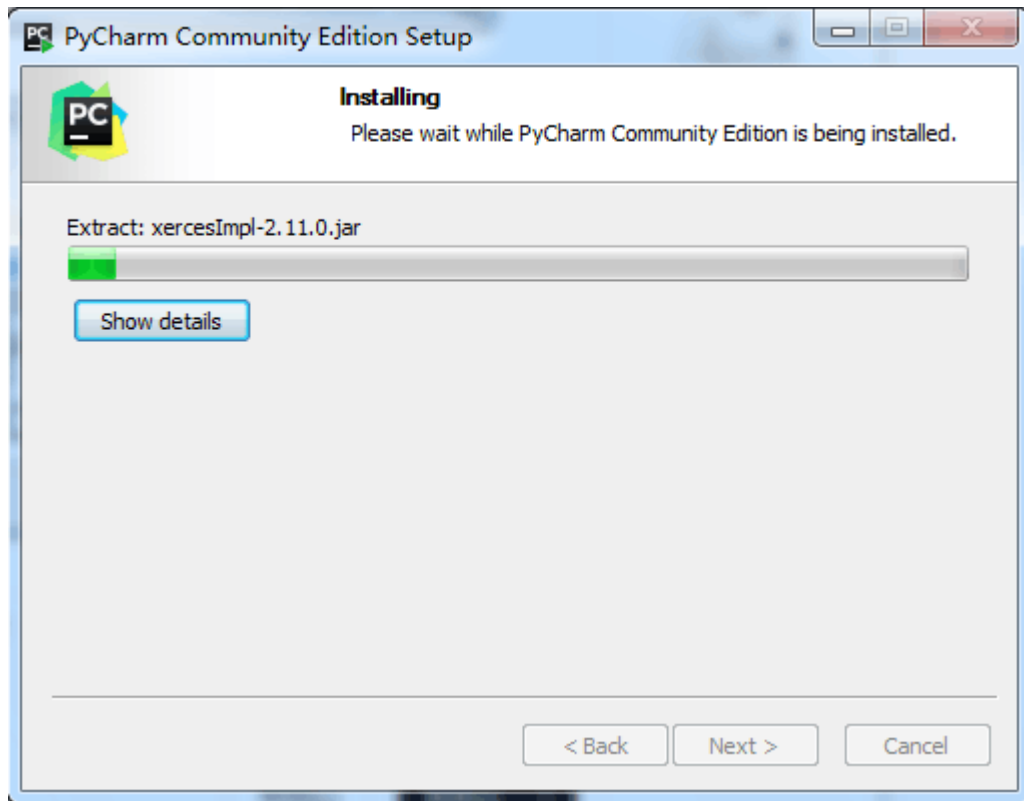


图 1-19 开始安装

(6) 安装完成后的界面如图 1-20 所示。左后点击【Install】按钮完成即可。



图 1-20 安装完成

1.3.2. PyCharm 的使用

完成 PyCharm 的安装后，就可以打开并使用 PyCharm 了。双击桌面的 PC 图标，首次使用 PyCharm 会提示用户接受安装协议，如图 1-21 所示。

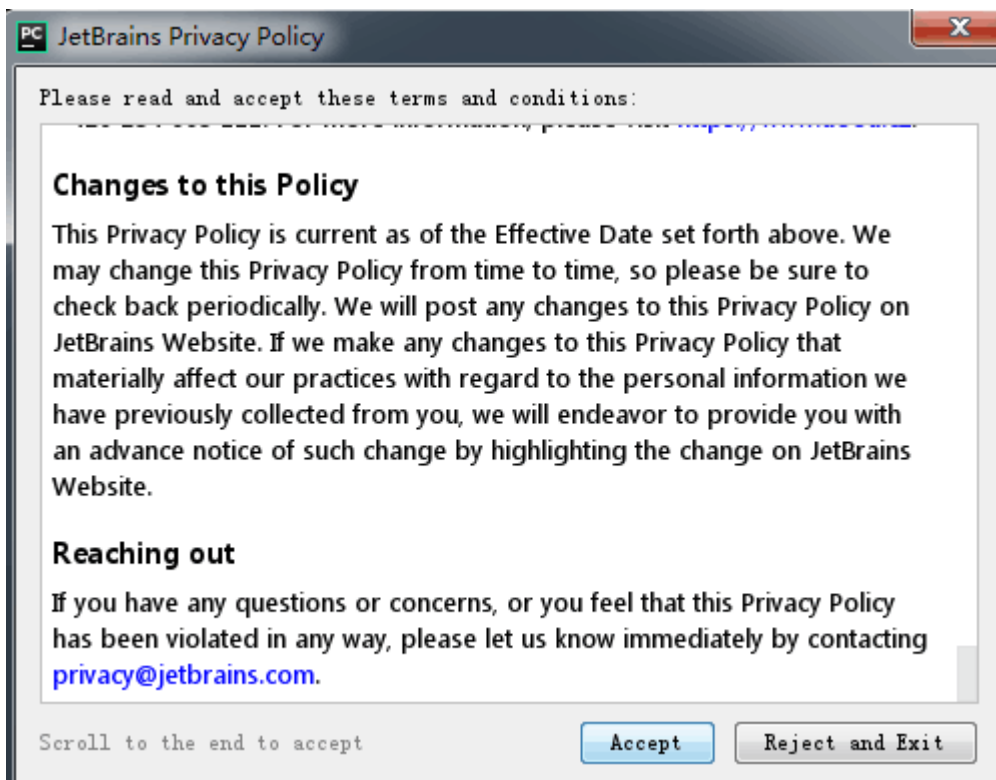


图 1-21 提示用户接受协议

单击图 1-21 中的【Accept】按钮，进入启动 PyCharm 界面，如图 1-22 所示。



图 1-22 启动 PyCharm

启动完成后，进入创建项目的界面，如图 1-23 所示。

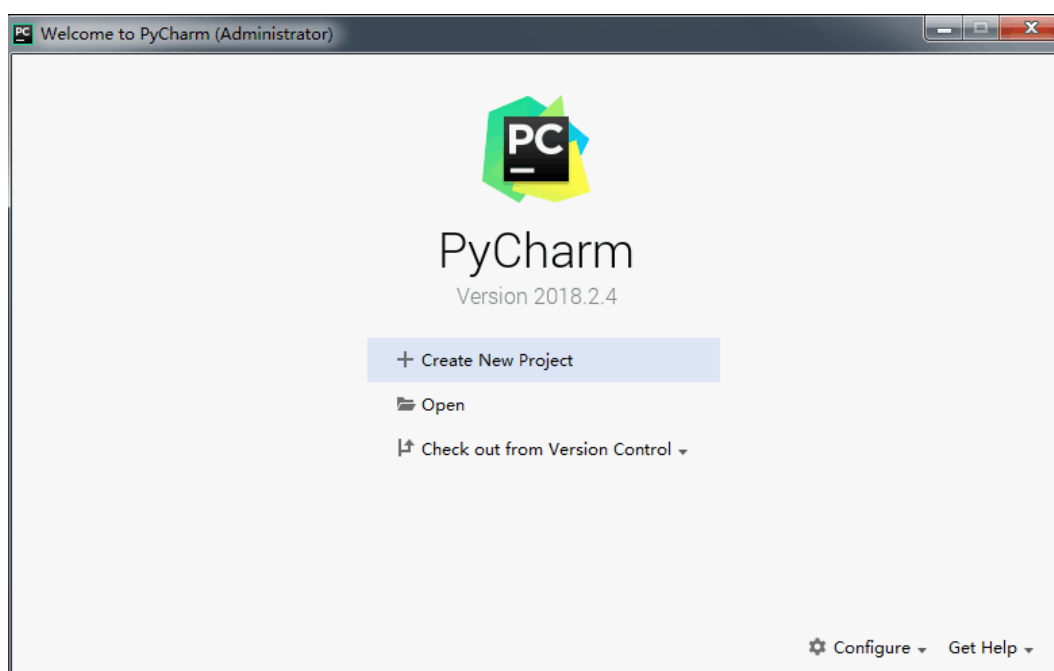


图 1-23 创建项目的界面

图 1-23 中共有三个选项，这三个选项的作用如下。

- (1) **【Create New Project】**: 用来创建一个新项目；
- (2) **【Open】**: 用来打开已经存在的项目
- (3) **【Check out from Version Control】**: 从版本控制中检出项目。

这里，我们选择第一个，创建一个新项目，单击**【Create New Project】**进入项目设置界面，如图 1-24 所示。

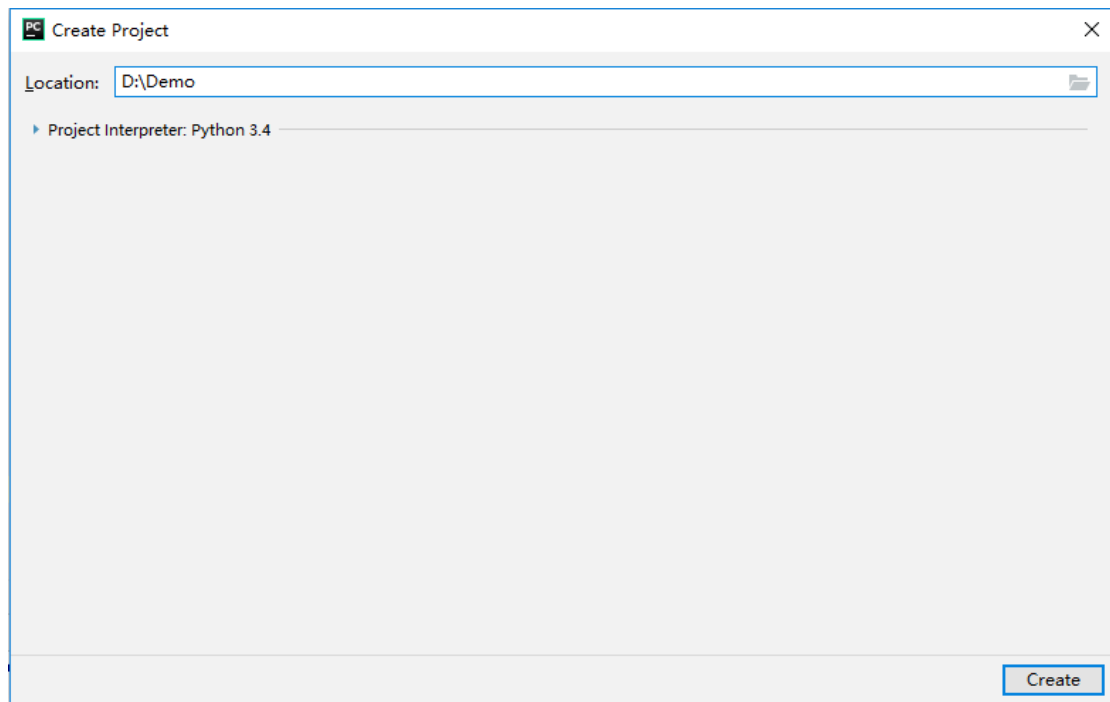


图 1-24 设置项目存放位置

假设，我们要将项目代码放在 D:\ Demo，设置好项目存放路径后，单击【Create】进入项目开发界面，如图 1-25 所示。

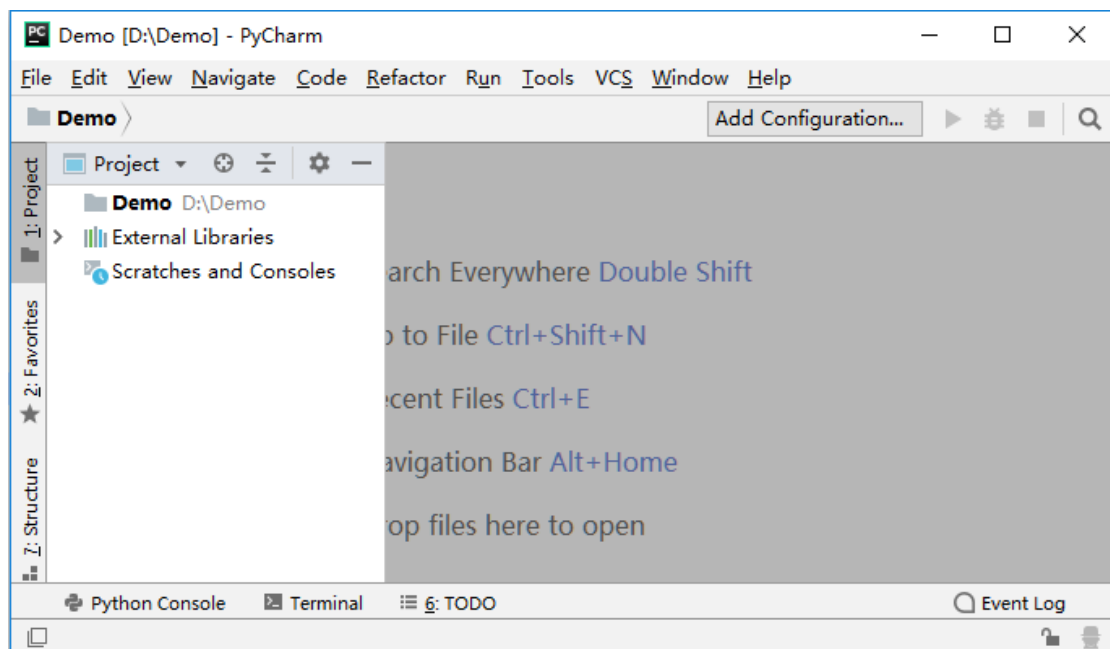


图 1-25 项目开发界面

创建好项目后，需要在项目中创建 Python 文件。选中项目名称，单击鼠标右键，在弹出的快捷菜单中选择【New】→【Python File】，如图 1-26 所示。

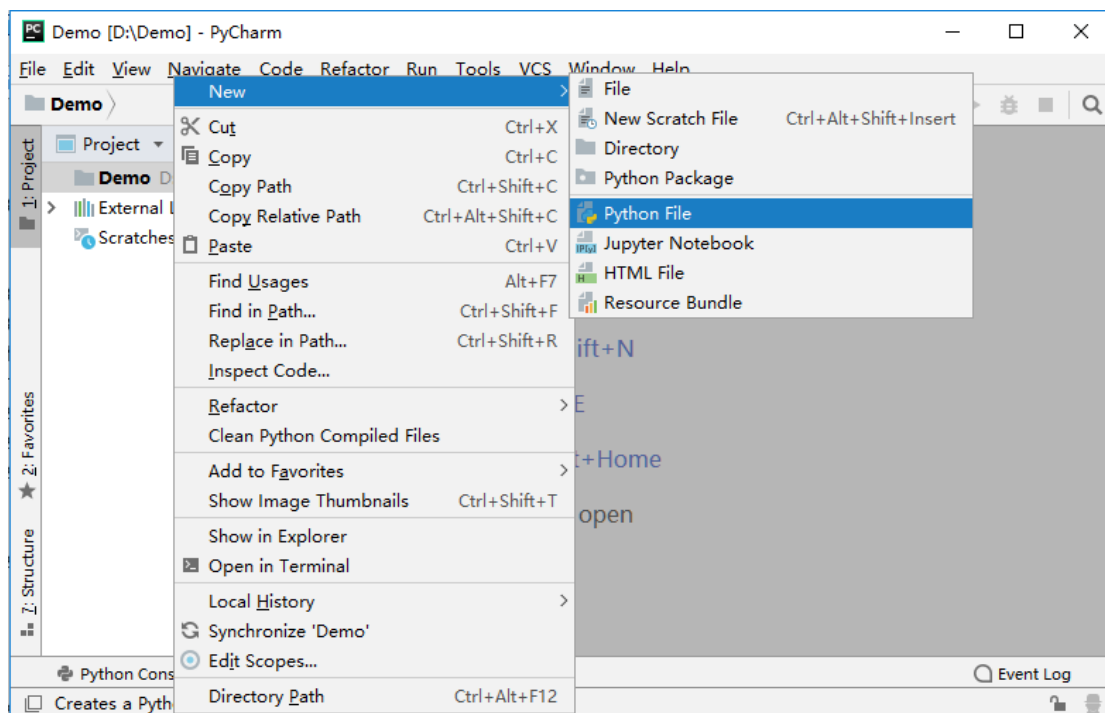


图 1-26 新建 Python 文件

为新建的 Python 文件命名，如图 1-27 所示。

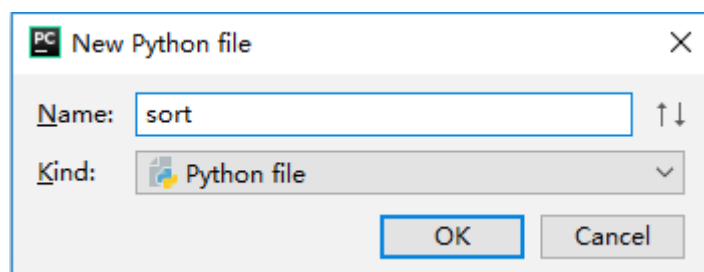


图 1-27 为 Python 文件命名

单击【OK】按钮后，创建好的文件界面如图 1-28 所示。

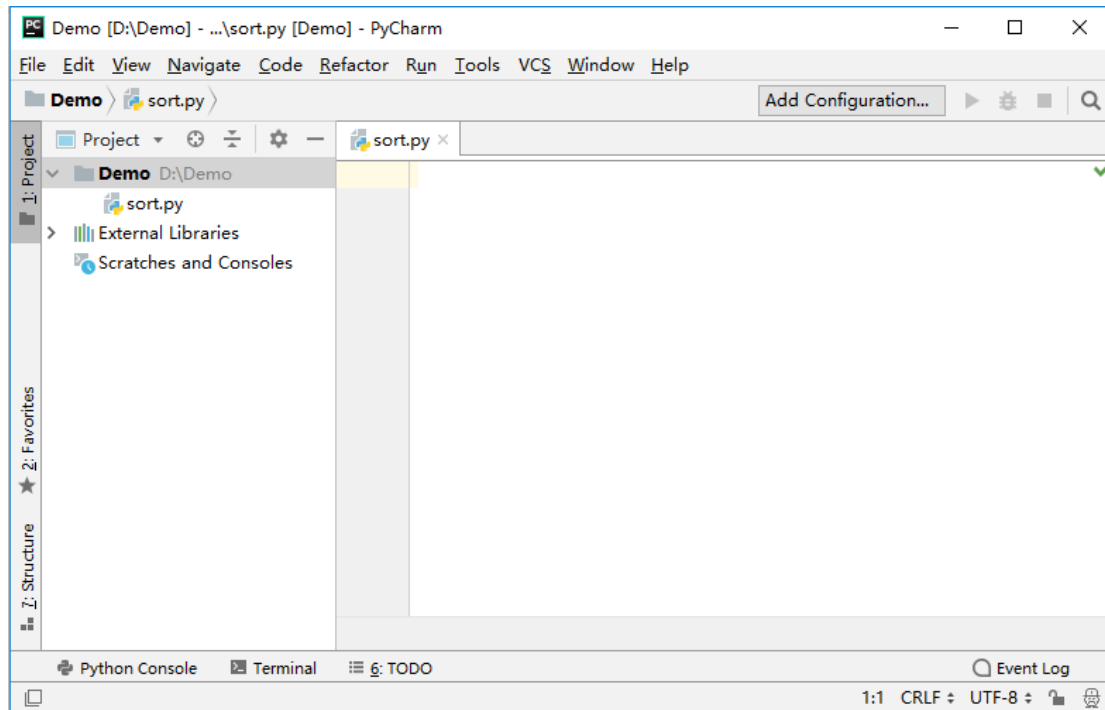


图 1-28 Python 文件创建好的界面

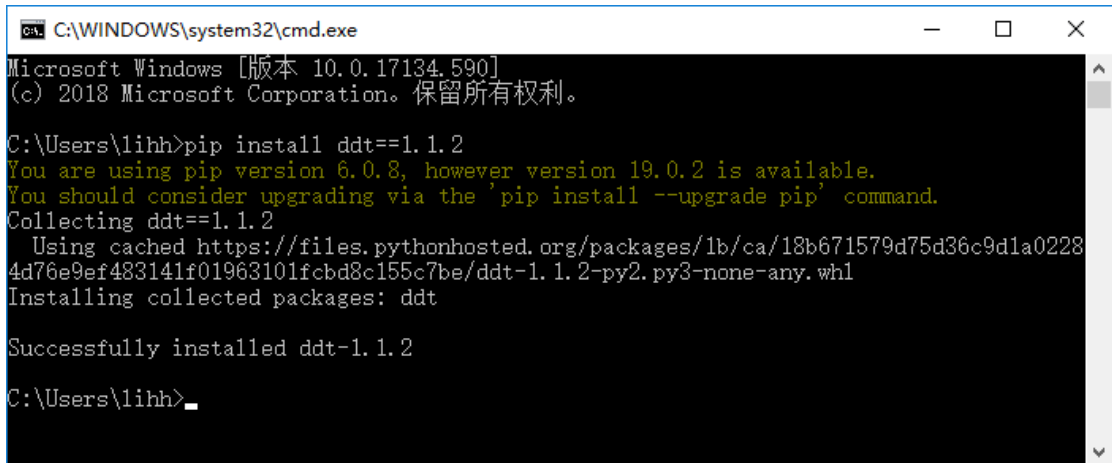
在创建好的 Python 文件中，可以加入被测代码，这里，我们在 sort 文件中输入如下函数：

```
def sort(num, type):  
    x = 0  
    y = 0  
    while num>0:  
        if type == 0:  
            x = y+2  
            break  
        elif type == 1:  
            x = y+10  
            break  
        else:  
            x = y+20  
            break  
    return x
```

1.4. 使用 ddt 创建数据驱动测试

1.4.1. 安装 ddt

使用 pip 命令进行安装，打开控制台，输入“pip install ddt==1.1.2”，控制台的输出结果如图 1-13 所示，此时，ddt 才被安装成功。



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.17134.590]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\lihh>pip install ddt==1.1.2
You are using pip version 6.0.8, however version 19.0.2 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
Collecting ddt==1.1.2
  Using cached https://files.pythonhosted.org/packages/1b/ca/18b671579d75d36c9d1a02284d76e9ef483141f01963101fcbd8c155c7be/ddt-1.1.2-py2.py3-none-any.whl
Installing collected packages: ddt
Successfully installed ddt-1.1.2
C:\Users\lihh>_
```

图 1-13 成功安装 ddt

1.4.2. 创建测试类

接下来开始使用 unittest 进行数据驱动测试，首先创建测试类，在项目下，新建一个 Python File，取名为“test_sort”，首先引入 unittest 模块、测试方法及 ddt 模块，然后定义一个类继承自 TestCase，为了创建数据驱动测试，我们需要在测试类上使用 @ddt 装饰符，具体如下：

```
import unittest

from sort import sort

from ddt import ddt,data,unpack

@ddt

class SortTestCase(unittest.TestCase):

    pass
```

所有的测试脚本都需要继承 `TestCase` 类，下面来写一个简单的测试用例。

1.4.3. `setUp()`方法

一个测试用例是从 `setUp()` 方法开始执行的，我们可以通过该方法在每个测试开始前去执行一些初始化的任务，这样做有助于确保每个测试方法都能够依赖相同的环境，无论类中有多少测试方法。

下面是添加 `setUp()` 方法的示例代码，在这个案例中，没有要进行初始化的操作，所以我们仅在 `setUp()` 方法中添加输出操作，具体如下。

```
def setUp(self):  
    print("test method start.....")
```

需要注意的是，`setUp()` 方法没有参数，而且不返回任何值

1.4.4. 编写测试

在之前的章节中，我们通过基本路径覆盖法，已经设计出 `sort` 方法的测试用例，如表 9-1 所示。

表 9-1 测试用例表

ID	输入数据	预期结果
测试用例 1	num=0 type=0	x=0
测试用例 2	num =1 type =0	x=2
测试用例 3	num =1 type =1	x=10
测试用例 4	num =1 type =2	x=20

现在我们将使用上述测试数据，为函数编写测试方法，我们需要给测试方法命名为 `test` 开头，这种命名约定通知 `test runner` 哪个方法代表测试方法，对于 `test runner` 能找到的每个测试方法，都会在执行测试方法之前先执行 `setUp()` 方法。这样做有助于确保每个测试方法都能够依赖相同的环境，无论类中有多少测试方法。

为 sort 方法添加测试方法 test_sort(), 并在测试方法上使用@data 装饰符, 具体代码如下。

```
@data([0,0,0],[1,0,2],[1,1,10],[1,2,20])
@unpack
def test_sort(self, x, y, expect_value):
    result = sort(x,y)
    self.assertEqual(result, expect_value, msg = result)
```

@data 装饰符可以把参数当作测试数据, 参数可以是单个值、列表、元组、字典。对于列表, 需要用@unpack 装饰符把元组和列表解析成多个参数。

在上面的代码里, test_sort()方法中, x、y 和 expect_value 三个参数用来接收元组解析的数据。

1.4.5. 代码清理

类似于 setUp() 方法, TestCase 类也会在测试执行完成之后调用 tearDown() 方法来清理所有的初始化值。

我们将在 tearDown() 方法中输出测试方法结束的标识, 示例代码如下:

```
def tearDown(self):
    print("test method end .....")
```

1.4.6. 运行测试

右键单击 sort_test 文件, 在弹出的快捷菜单中选择 **【Run ‘unittests in sort_test’】** 运行程序, 如图 1-29 所示。

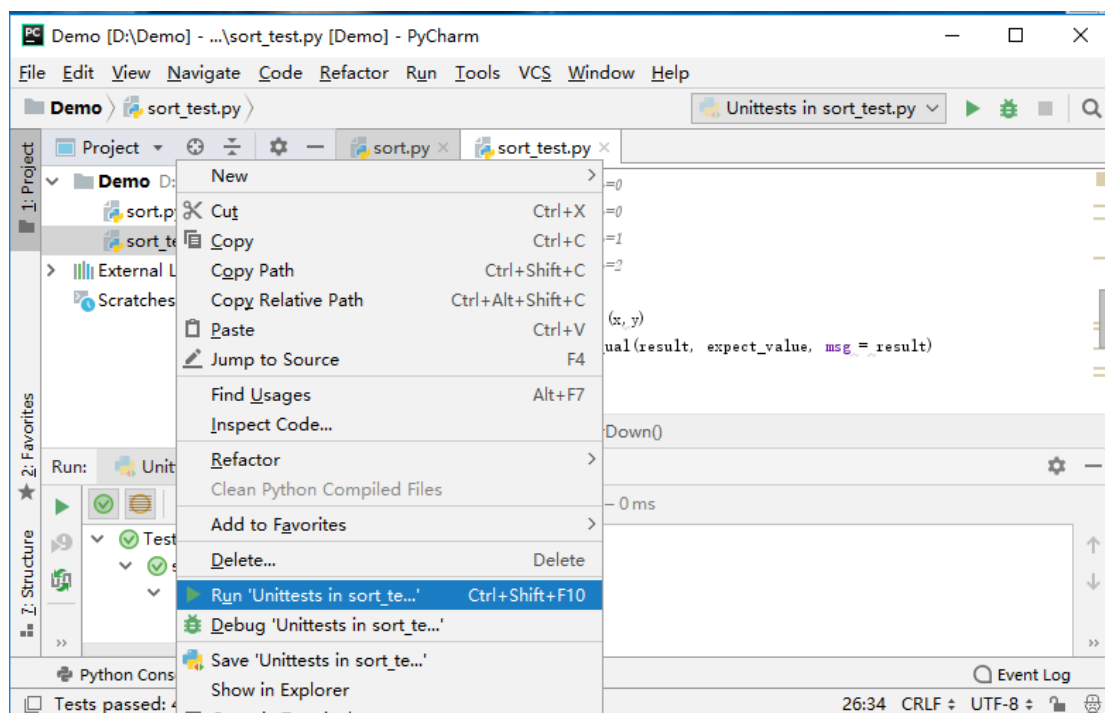


图 6-2 运行程序

为了通过命令行运行测试，我们可以在测试用例中添加对 main 方法的调用，示例代码如下：

```
if __name__ == '__main__':  
    unittest.main(verbosity=2)
```

此时，在脚本所在路径，输入 “python -m sort_test”，运行测试脚本的时候，ddt 把测试数据转换为有效的 python 标识符，生成名称更有意义的测试方法，例如上面的测试，ddt 将生成如图 6-3 展示的方法名。


```
C:\WINDOWS\system32\cmd.exe

D:\Demo>python -m sort_test
test_sort_1_0_0_0_ (__main__.SortTestCase) ... test method start .....
>
test method end .....
ok
test_sort_2_1_0_2_ (__main__.SortTestCase) ... test method start .....
>
test method end .....
ok
test_sort_3_1_1_10_ (__main__.SortTestCase) ... test method start ...
...>
test method end .....
ok
test_sort_4_1_2_20_ (__main__.SortTestCase) ... test method start ...
...>
test method end .....
ok

-----
Ran 4 tests in 0.005s
```

图 6-3 控制台运行程序

1.5. 断言

unittest 的 TestCase 类提供了一些方法来校验预期结果和程序返回的实际结果是否一致，下面列出了一些基本断言，如表 6-1 所示。

表 6-1 基本断言方法列表

方 法	校 验 条 件	应用实例
assertEqual (a, b [,msg])	a == b	这些方法校验a和b是否相等，msg对象是用来说明失败原因的消息。 这对于验证元素的值和属性等是非常有用的，例如： assertEqual(element.text, "10")
assertNotEqual(a, b[,msg])	a != b	
assertTrue(x[,msg]))	bool(x) is True	这些方法校验给出的表达式是True还是False。 例如，校验一个元素是否出现在页面，我们可以用下面的方法： assertTrue(element.is_displayed())
assertFalse(x[,msg]))	bool(x) is False	

<code>assertIn(a, b,[msg])</code>	<code>a in b</code>	这些方法验证b是否包含a，msg对象是用来说明失败原因的消息。
<code>assertNotIn(a, b,[msg])</code>	<code>a not in b</code>	
<code>assertAlmostEqual(a, b)</code>	<code>round(a-b, 7) == 0</code>	这些方法用于检查数值，在检查之前会按照给定的精度把数字四舍五入。这有助于统计由于四舍五入产生的错误和其他由于浮点运算产生的问题
<code>assertNotAlmostEqual(a, b)</code>	<code>round(a-b, 7) != 0</code>	
<code>assertGreater(a, b)</code>	<code>a > b</code>	这些方法类似于 <code>assertEqual()</code> 方法，是为逻辑判定条件设计的
<code>assertGreaterEqual(a, b)</code>	<code>a >= b</code>	
<code>assertLess(a, b)</code>	<code>a < b</code>	
<code>assertLessEqual(a, b)</code>	<code>a <= b</code>	
<code>assertRegexMatches(s, r)</code>	<code>r.search(s)</code>	这些方法检查文本是否符合正则匹配
<code>assertNotRegexMatches(s, r)</code>	<code>not r.search(s)</code>	
<code>assertListEqual(a, b)</code>	<code>lists</code>	此方法校验两个list是否相等，对于下拉列表选项字段的校验是非常有用的
<code>fail()</code>		此方法是无条件的失败。在别的assert 方法不好用的时候，也可用此方法来创建定制的条件块

右键单击 `sort_login` 文件，在弹出的快捷菜单中选择 【Run ‘unittests in sort_test’】运行程序，如果成功，断言方法则标识该测试为成功状态，如图 6-4 所示。

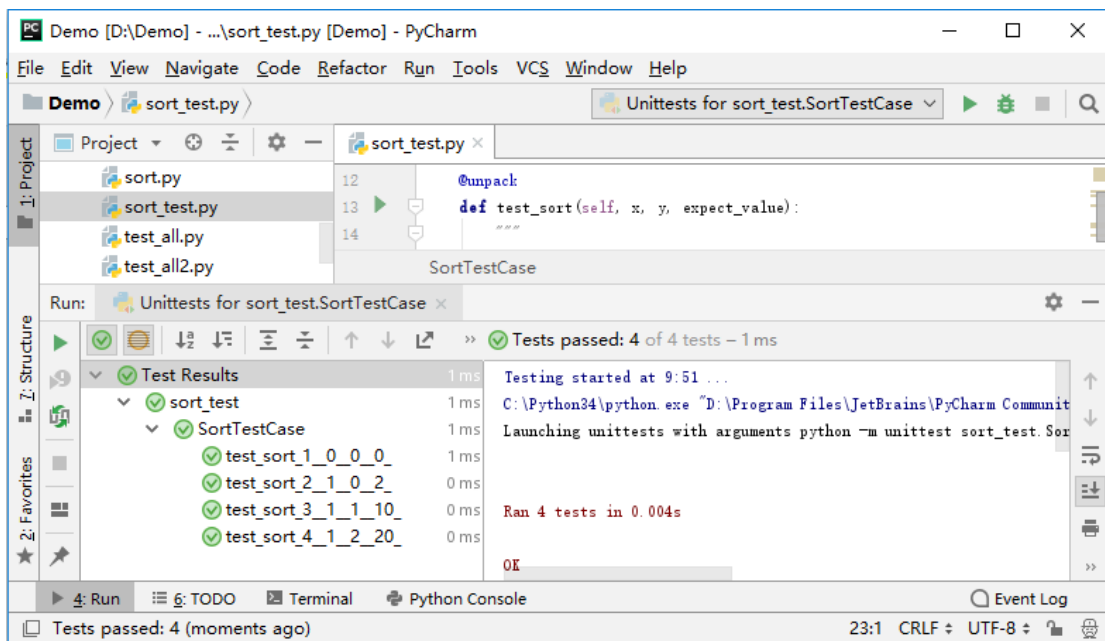


图 6-4 unittest 断言成功

如果断言失败，则抛出一个 `AssertionError`，并标识该测试为失败状态，针对每个失败，测试结果概要都会通过生成文本信息来展示具体哪里有错误，修改预期结果的值，如图 6-5 所示。

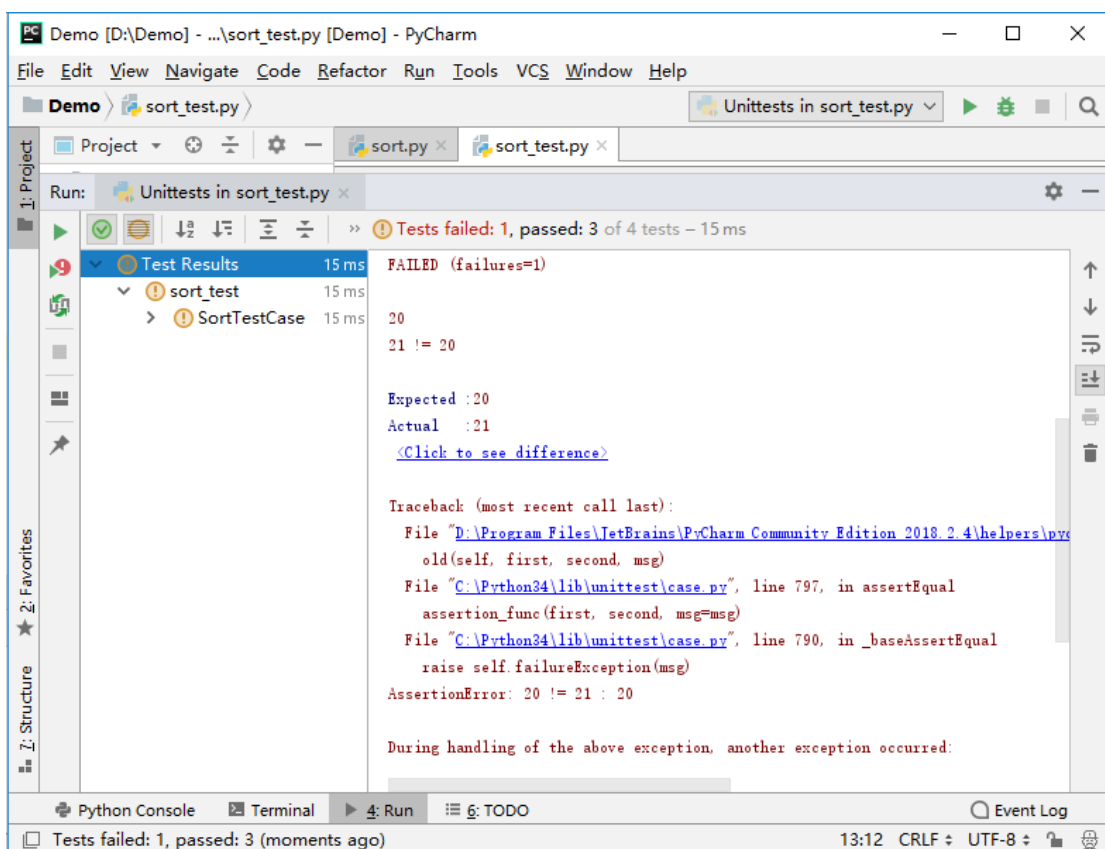


图 6-5 unittest 断言失败结果展示

1.6. 测试套件

只有一个测试文件，直接执行该文件即可，但若有多个测试文件，我们就需要组织测试、批量执行的问题。

在了解 TestSuites 的细节之前，我们要添加一个新的测试。

(1) 新建一个 Python File，取名“abs”，作为被测程序，具体如下：

```
def abs(n):
    if n > 0:
        return n
    elif n < 0:
        return -n
    else:
        return 0
```

(2) 为 abs 函数添加一个测试类，具体代码如下：

```
import unittest
from abs import abs
from ddt import ddt, data, unpack

@ddt
class AbsTestCase(unittest.TestCase):

    def setUp(self):
        print("test method start .....>")

    @data([-1, 1], [1, 1], [0, 0])
    @unpack
    def test_abs(self, n, expect_value):
        result = abs(n)
        self.assertEqual(result, expect_value, msg = result)

    def tearDown(self):
        print("test method end .....")

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

右键单击 abs_test 文件，在弹出的快捷菜单中选择【Run ‘unittests in abs_test’】运行程序，如果成功，断言方法则标识该测试为成功状态，如图 6-7 所示。

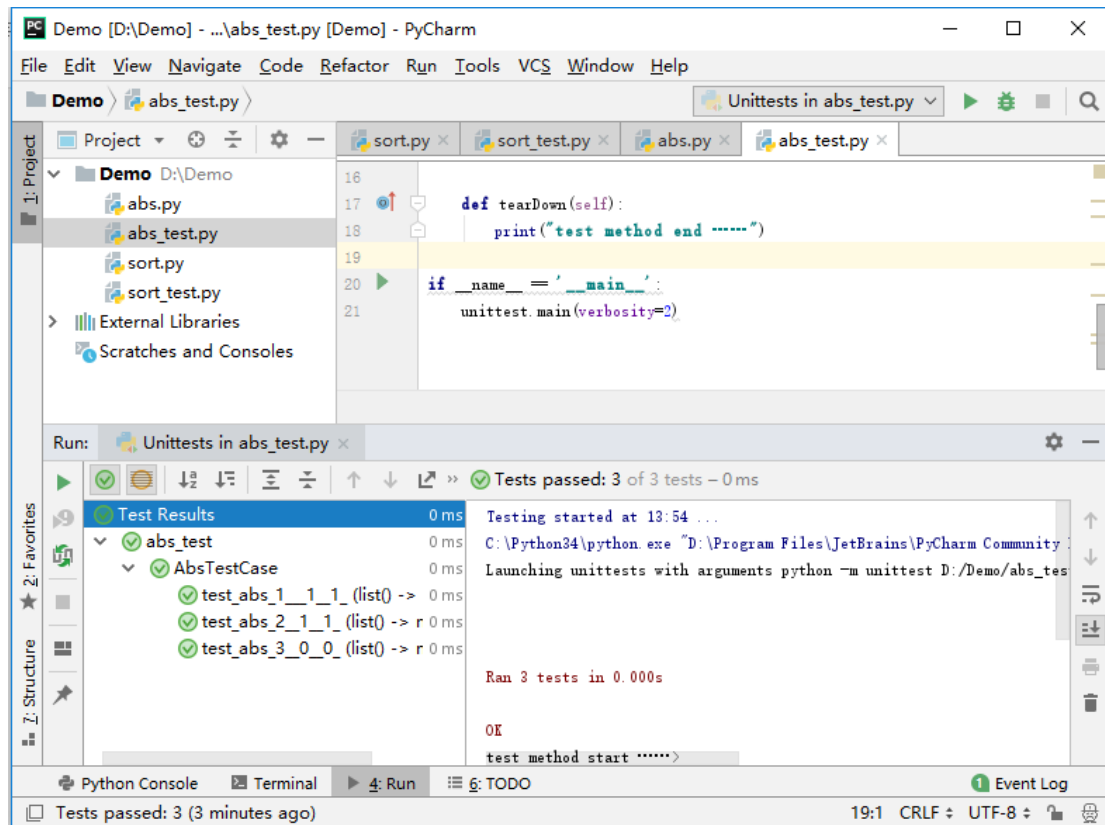


图 6-7 测试执行通过

1.6.1. TestSuite 执行指定用例

我们使用测试套件 TestSuite 来组织和运行测试。

(1) 新建一个 Python File，取名“run_all”，首先引入 unittest 模块，导入要运行的测试类，具体如下：

```
import unittest

from abs_test import AbsTestCase

from sort_test import SortTestCase
```

(2) 创建一个测试套件实例，利用 makeSuite() 方法，一次性加载一个类文件下所有测试用例到 suite 中去，具体代码如下。

```
# 构造测试套件
```

```
suite = unittest.TestSuite()

suite.addTest(unittest.makeSuite(AbsTestCase))

suite.addTest(unittest.makeSuite(SortTestCase))
```

(3) TestRunner 类通过 run 方法调用测试套件来执行文件中所有的测试，具体如下：

```
if __name__ == "__main__":
    # 执行测试

    runner = unittest.TextTestRunner(verbosity=2)

    runner.run(suite)
```

注意：verbosity 参数可以控制输出的错误报告的详细程度，默认是 1，如果设为 0，则不输出每一用例的执行结果；如果设为 2，则输出详细的执行结果。

(4) 右键单击 test_all 文件，在弹出的菜单中选择【Run ‘test_all’】运行程序，如图 6-8 所示。

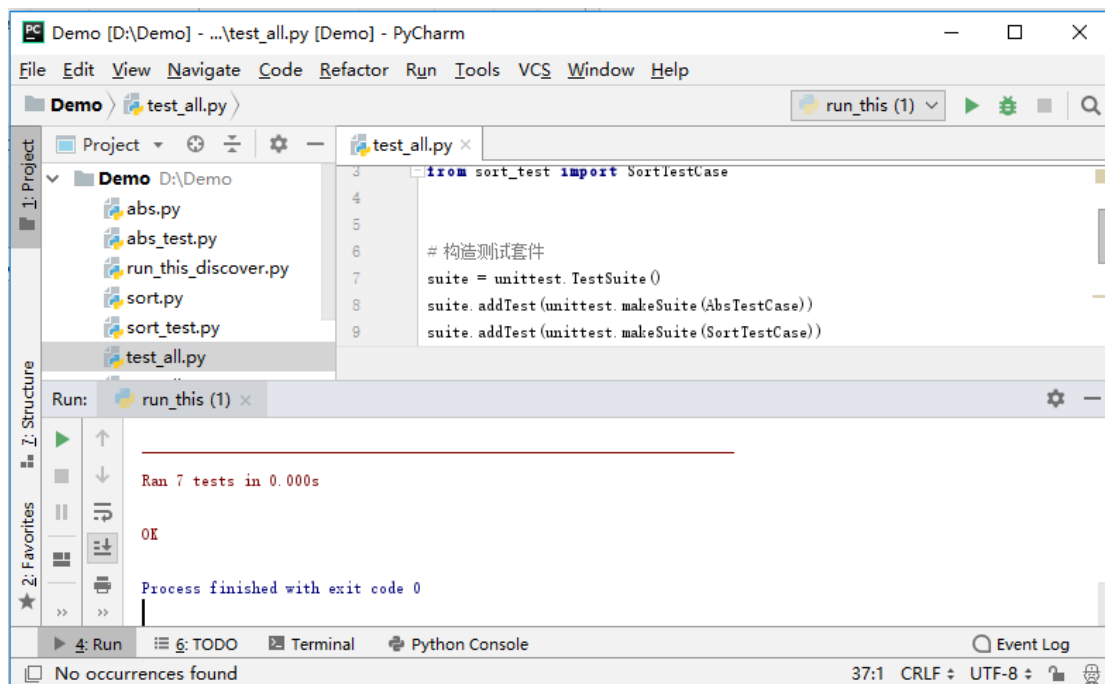
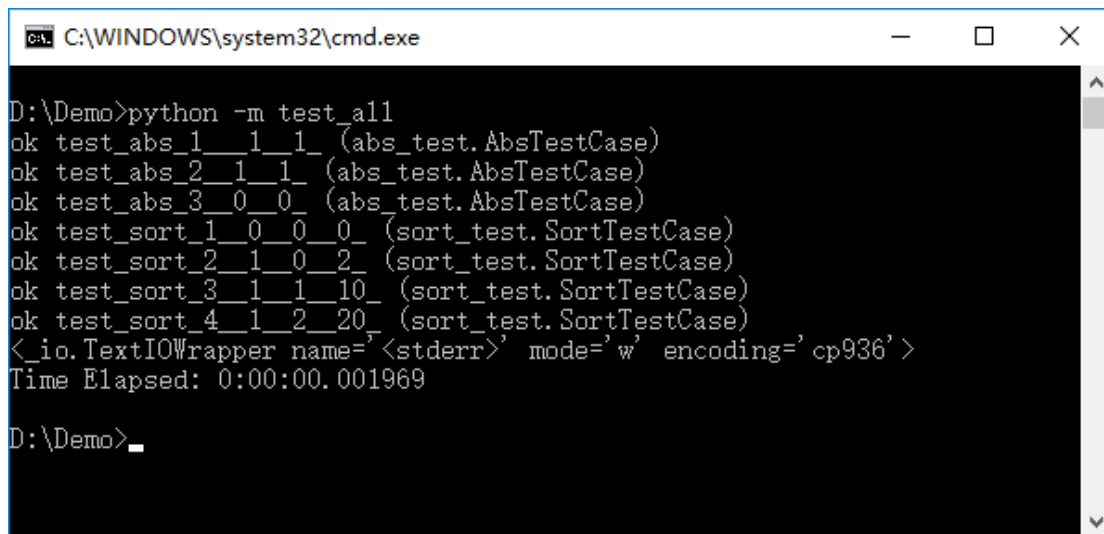


图 6-8 多测试执行结果

我们也可以在控制台输入 “python -m test_all”，控制台会输出测试结果，如图 6-9 所示。



```
C:\WINDOWS\system32\cmd.exe

D:\Demo>python -m test_all
ok test_abs_1_1_1_1_ (abs_test.AbsTestCase)
ok test_abs_2_1_1_1_ (abs_test.AbsTestCase)
ok test_abs_3_0_0_0_ (abs_test.AbsTestCase)
ok test_sort_1_0_0_0_ (sort_test.SortTestCase)
ok test_sort_2_1_0_2_ (sort_test.SortTestCase)
ok test_sort_3_1_1_10_ (sort_test.SortTestCase)
ok test_sort_4_1_2_20_ (sort_test.SortTestCase)
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='cp936'>
Time Elapsed: 0:00:00.001969

D:\Demo>_
```

图 6-9 多测试控制台执行结果

1.6.2. discover 批量加载用例

当有上百个用例文件时，使用列表单个加入用例效率比较低，我们可以通过 unittest 的 discover() 方法批量加载用例。

(1) 新建一个 Python File，取名 “run_this_discover”，首先导入 unittest 模块：

```
import unittest
```

(2) 通过 discover 方法批量加载测试，具体代码如下：

```
# 测试用例目录
test_dir = './'

# 加载测试用例
suite = unittest.TestLoader().discover(test_dir, pattern='test*.py')
```

discover 方法可以匹配某个目录下，符合某种规则的用例文件：

```
discover(start_dir, pattern='*test.py', top_level_dir=None)
```

- start_dir: 测试用例所在目录
- pattern='*test.py': 表示用例文件名的匹配方式，此处匹配的是以

test 结尾的.py 类型的文件，*表示匹配任意字符

- top_level_dir: 测试模块的顶层目录

(3) TestRunner 类通过 run 方法调用测试套件来执行文件中所有的测试，具体如下：

```
if __name__ == "__main__":  
    # 执行测试  
  
    runner = unittest.TextTestRunner(verbosity=2)  
  
    runner.run(suite)
```

运行程序，结果如图 6-10 所示。

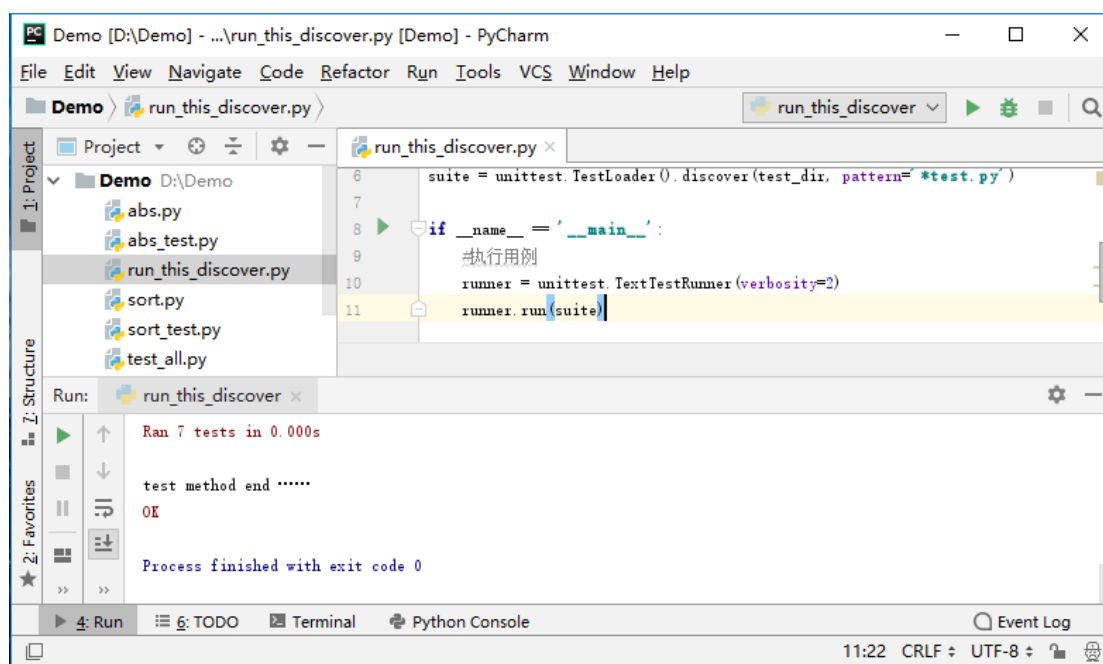


图 6-10 多测试执行结果

1.7. 生成 HTML 格式的测试报告

最后我们需要生成测试报告，unittest 没有相应的内置模块可以生成格式友好的报告，我们可以应用 unittest 的扩展 HTMLTestRunner 来实现。

(1)访问 <https://pypi.python.org/pypi/HTMLTestRunner>，下载该扩展，如图 6-11 所示，并放在项目根目录下。

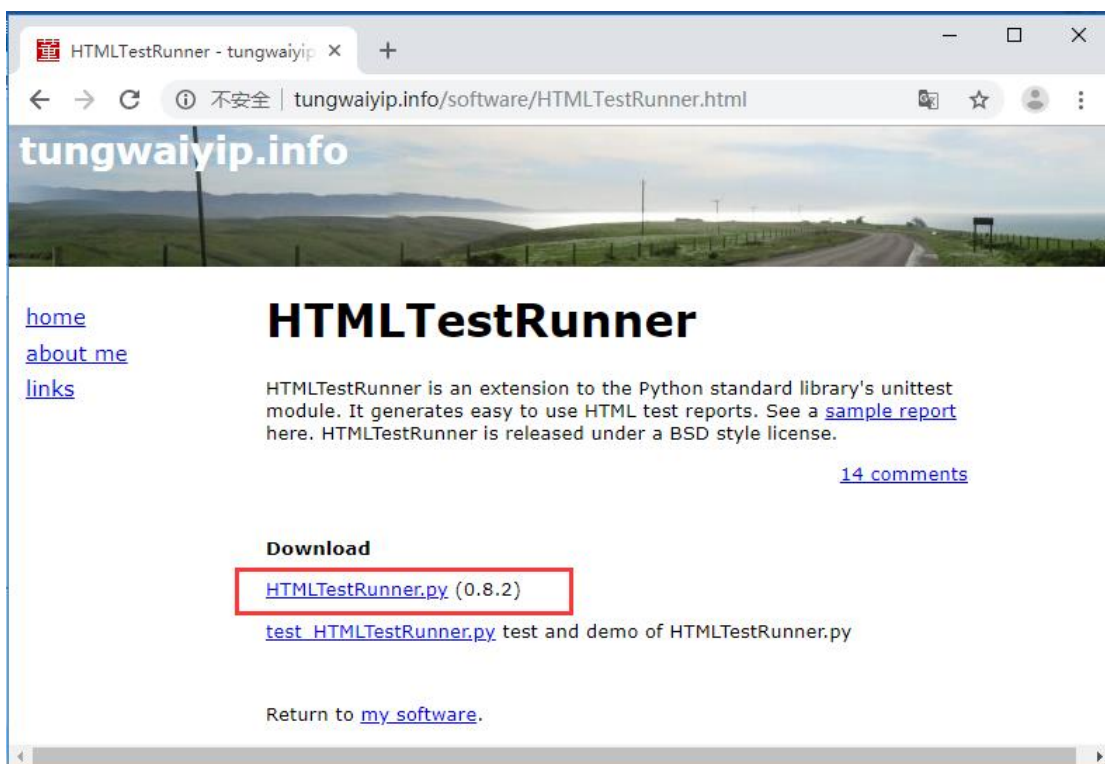


图 6-11 HTMLTestRunner 下载页面

(2) HTMLTestRunner 用 python2 编写，我们使用的是 python3，因此需要修改该文件，具体如下：

```
94 行: import StringIO 修改成 import io
539 行: self.outputBuffer = StringIO.StringIO() 修改成 self.outputBuffer =
io.StringIO()
642 行: if not rmap.has_key(cls) 修改成 if not cls in rmap:
766 行: uo = o.decode('latin-1') 修改成 uo = e
772 行: ue = e.decode('latin-1') 修改成 ue = e
631 行: print >> sys.stderr, '\nTime Elapsed: %s' % (self.stopTime-self.startTime)修改成
print(sys.stderr, '\nTime Elapsed: %s' % (self.stopTime-self.startTime))
```

(3) 新建一个 Directory，取名“report”，作为测试报告的存放目录。

(4) 修改“test_all”文件，首先导入 HTMLTestRunner 及 os 模块

```
import HTMLTestRunner
import os
```

(5) 设置报告文件的保存路径，具体代码如下。

```
# 设置报告文件保存路径

cur_path = os.path.dirname(os.path.realpath(__file__))
report_path = os.path.join(cur_path, "report")
if not os.path.exists(report_path): os.mkdir(report_path)
```

(6) 构造测试套件，这个大家已经比较熟悉了，具体如下。

```
# 构造测试套件

suite = unittest.TestSuite()

suite.addTest(unittest.makeSuite(AbsTestCase))

suite.addTest(unittest.makeSuite(SortTestCase))
```

(7) 构造 HTMLTestRunner 实例，通过 run 方法调用测试套件来执行文件中所有的测试，具体如下：

```
if __name__ == "__main__":

    # 打开一个文件，将 result 写入此 file 中

    html_report = report_path + r"\result.html"

    fp = open(html_report, "wb")

    # 初始化一个 HTMLTestRunner 实例对象，用来生成报告

    runner = HTMLTestRunner.HTMLTestRunner(stream=fp, verbosity=2,

                                              title="单元测试报告", description="用例执行情况")

    runner.run(suite)
```

右键单击 run_this 文件，在弹出的菜单中选择【Run ‘test_all’】运行程序，查看测试报告，如图 6-12 所示。

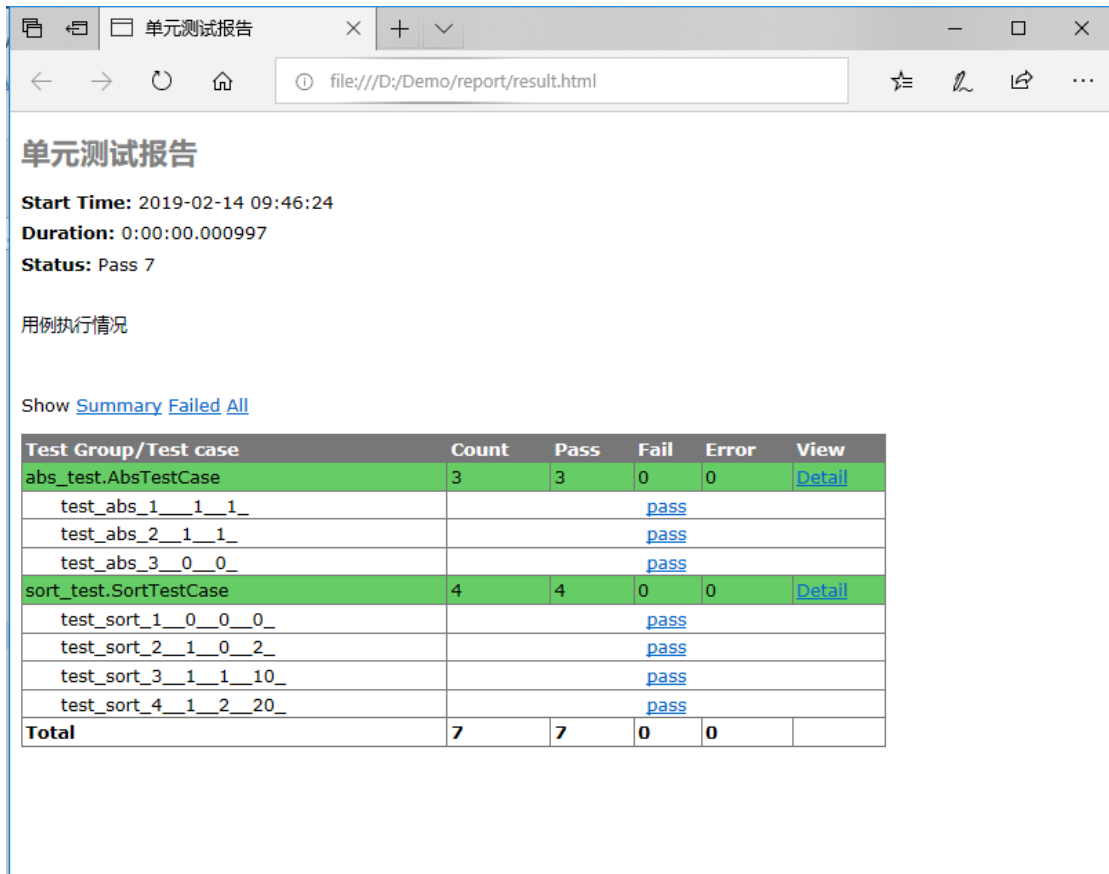


图 6-12 测试报告

1.1. 本章小结

Unittest 是 python 自带的测试框架，主要适用于单元测试，可以对多个测试用例进行管理和封装，并通过执行输出测试结果，本章主要介绍该框架在进行单元测试时的使用过程。

在通过白盒测试方法进行测试用例设计后，可以通过 unittest 来完成整个单元测试的构建，编写测试、批量运行及最终生成测试报告，实际企业中还会与 jenkins 结合，在后续章节中会介绍持续集成工具 jenkins，实现持续集成与自动化测试的目标。