本文将讨论Python中下划线（\_）字符的使用方法。我们将会看到，正如Python中的很多事情，下划线的不同用法大多数（并非所有）本文将讨论Python中下划线（\_）字符的使用方法。我们将会看到，正如Python中的很多事情，下划线的不同用法大多数（并非所有）只是常用惯例而已。

单下划线（\_）

通常情况下，会在以下3种场景中使用：

1、在解释器中：在这种情况下，“\_”代表交互式解释器会话中上一条执行的语句的结果。这种用法首先被标准CPython解释器采用，然后其他类型的解释器也先后采用。

Python

>>> \_ Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name '\_' is not defined

>>> 42

>>> \_

42

>>> 'alright!' if \_ else ':('

'alright!'

>>> \_

'alright!'

1

2

3

4

5

6

7

8

9

10

>>> \_ Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name '\_' is not defined

>>> 42

>>> \_

42

>>> 'alright!' if \_ else ':('

'alright!'

>>> \_

'alright!'

2、作为一个名称：这与上面一点稍微有些联系，此时“\_”作为临时性的名称使用。这样，当其他人阅读你的代码时将会知道，你分配了一个特定的名称，但是并不会在后面再次用到该名称。例如，下面的例子中，你可能对循环计数中的实际值并不感兴趣，此时就可以使用“\_”。

Python

n = 42

for \_ in range(n):

do\_something()

1

2

3

n = 42

for \_ in range(n):

do\_something()

3、国际化：也许你也曾看到”\_“会被作为一个函数来使用。这种情况下，它通常用于实现国际化和本地化字符串之间翻译查找的函数名称，这似乎源自并遵循相应的C约定。例如，在Django文档“转换”章节中，你将能看到如下代码：

Python

from django.utils.translation import ugettext as \_

from django.http import HttpResponse

def my\_view(request):

output = \_("Welcome to my site.")

return HttpResponse(output)

1

2

3

4

5

from django.utils.translation import ugettext as \_

from django.http import HttpResponse

def my\_view(request):

output = \_("Welcome to my site.")

return HttpResponse(output)

可以发现，场景二和场景三中的使用方法可能会相互冲突，所以我们需要避免在使用“\_”作为国际化查找转换功能的代码块中同时使用“\_”作为临时名称。

名称前的单下划线（如：\_shahriar）

程序员使用名称前的单下划线，用于指定该名称属性为“私有”。这有点类似于惯例，为了使其他人（或你自己）使用这些代码时将会知道以“\_”开头的名称只供内部使用。正如Python文档中所述：

以下划线“\_”为前缀的名称（如\_spam）应该被视为API中非公开的部分（不管是函数、方法还是数据成员）。此时，应该将它们看作是一种实现细节，在修改它们时无需对外部通知。

正如上面所说，这确实类似一种惯例，因为它对解释器来说确实有一定的意义，如果你写了代码“from <模块/包名> import \*”，那么以“\_”开头的名称都不会被导入，除非模块或包中的“\_\_all\_\_”列表显式地包含了它们。了解更多请查看“Importing \* in Python”。

名称前的双下划线（如：\_\_shahriar）

名称（具体为一个方法名）前双下划线（\_\_）的用法并不是一种惯例，对解释器来说它有特定的意义。Python中的这种用法是为了避免与子类定义的名称冲突。Python文档指出，“\_\_spam”这种形式（至少两个前导下划线，最多一个后续下划线）的任何标识符将会被“\_classname\_\_spam”这种形式原文取代，在这里“classname”是去掉前导下划线的当前类名。例如下面的例子：

Python

>>> class A(object):

... def \_internal\_use(self):

... pass

... def \_\_method\_name(self):

... pass

...

>>> dir(A())

['\_A\_\_method\_name', ..., '\_internal\_use']

1

2

3

4

5

6

7

8

>>> class A(object):

... def \_internal\_use(self):

... pass

... def \_\_method\_name(self):

... pass

...

>>> dir(A())

['\_A\_\_method\_name', ..., '\_internal\_use']

正如所预料的，“\_internal\_use”并未改变，而“\_\_method\_name”却被变成了“\_ClassName\_\_method\_name”。此时，如果你创建A的一个子类B，那么你将不能轻易地覆写A中的方法“\_\_method\_name”。

Python

>>> class B(A):

... def \_\_method\_name(self):

... pass

...

>>> dir(B())

['\_A\_\_method\_name', '\_B\_\_method\_name', ..., '\_internal\_use']

1

2

3

4

5

6

>>> class B(A):

... def \_\_method\_name(self):

... pass

...

>>> dir(B())

['\_A\_\_method\_name', '\_B\_\_method\_name', ..., '\_internal\_use']

这里的功能几乎和Java中的final方法和C++类中标准方法（非虚方法）一样。

名称前后的双下划线（如：\_\_init\_\_）

这种用法表示Python中特殊的方法名。其实，这只是一种惯例，对Python系统来说，这将确保不会与用户自定义的名称冲突。通常，你将会覆写这些方法，并在里面实现你所需要的功能，以便Python调用它们。例如，当定义一个类时，你经常会覆写“\_\_init\_\_”方法。

虽然你也可以编写自己的特殊方法名，但不要这样做。

Python

>>> class C(object):

... def \_\_mine\_\_(self):

... pass

...

>>> dir(C)

... [..., '\_\_mine\_\_', ...]

1

2

3

4

5

6

>>> class C(object):

... def \_\_mine\_\_(self):

... pass

...

>>> dir(C)

... [..., '\_\_mine\_\_', ...]

其实，很容易摆脱这种类型的命名，而只让Python内部定义的特殊名称遵循这种约定只是常用惯例而已。

**单下划线（\_）**

通常情况下，会在以下3种场景中使用：

1、**在解释器中**：在这种情况下，“\_”代表交互式解释器会话中上一条执行的语句的结果。这种用法首先被标准CPython解释器采用，然后其他类型的解释器也先后采用。

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | >>> \_ Traceback (most recent call last):  File "<stdin>", line 1, in <module>  NameError: name '\_' is not defined  >>> 42  >>> \_  42  >>> 'alright!' if \_ else ':('  'alright!'  >>> \_  'alright!' |

2、**作为一个名称**：这与上面一点稍微有些联系，此时“\_”作为临时性的名称使用。这样，当其他人阅读你的代码时将会知道，你分配了一个特定的名称，但是并不会在后面再次用到该名称。例如，下面的例子中，你可能对循环计数中的实际值并不感兴趣，此时就可以使用“\_”。

Python



|  |  |
| --- | --- |
| 1  2  3 | n = 42  for \_ in range(n):      do\_something() |

3、**国际化**：也许你也曾看到”\_“会被作为一个函数来使用。这种情况下，它通常用于实现国际化和本地化字符串之间翻译查找的函数名称，这似乎源自并遵循相应的C约定。例如，在[Django文档“转换”章节](https://docs.djangoproject.com/en/dev/topics/i18n/translation/)中，你将能看到如下代码：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5 | from django.utils.translation import ugettext as \_  from django.http import HttpResponse  def my\_view(request):  output = \_("Welcome to my site.")  return HttpResponse(output) |

可以发现，场景二和场景三中的使用方法可能会相互冲突，所以我们需要避免在使用“\_”作为国际化查找转换功能的代码块中同时使用“\_”作为临时名称。

**名称前的单下划线（如：\_shahriar）**

程序员使用名称前的单下划线，用于指定该名称属性为“私有”。这有点类似于惯例，为了使其他人（或你自己）使用这些代码时将会知道以“\_”开头的名称只供内部使用。正如Python文档中所述：

以下划线“\_”为前缀的名称（如\_spam）应该被视为API中非公开的部分（不管是函数、方法还是数据成员）。此时，应该将它们看作是一种实现细节，在修改它们时无需对外部通知。

正如上面所说，这确实类似一种惯例，因为它对解释器来说确实有一定的意义，如果你写了代码“from <模块/包名> import \*”，那么以“\_”开头的名称都不会被导入，除非模块或包中的“\_\_all\_\_”列表显式地包含了它们。了解更多请查看“[Importing \* in Python](http://shahriar.svbtle.com/importing-star-in-python)”。

**名称前的双下划线（如：\_\_shahriar）**

名称（具体为一个方法名）前双下划线（\_\_）的用法并不是一种惯例，对解释器来说它有特定的意义。Python中的这种用法是为了避免与子类定义的名称冲突。Python文档指出，“\_\_spam”这种形式（至少两个前导下划线，最多一个后续下划线）的任何标识符将会被“\_classname\_\_spam”这种形式原文取代，在这里“classname”是去掉前导下划线的当前类名。例如下面的例子：

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> class A(object):  ... def \_internal\_use(self):  ... pass  ... def \_\_method\_name(self):  ... pass  ...  >>> dir(A())  ['\_A\_\_method\_name', ..., '\_internal\_use'] |

正如所预料的，“\_internal\_use”并未改变，而“\_\_method\_name”却被变成了“\_ClassName\_\_method\_name”。此时，如果你创建A的一个子类B，那么你将不能轻易地覆写A中的方法“\_\_method\_name”。

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> class B(A):  ... def \_\_method\_name(self):  ... pass  ...  >>> dir(B())  ['\_A\_\_method\_name', '\_B\_\_method\_name', ..., '\_internal\_use'] |

这里的功能几乎和Java中的final方法和C++类中标准方法（非虚方法）一样。

**名称前后的双下划线（如：\_\_init\_\_）**

这种用法表示Python中特殊的方法名。其实，这只是一种惯例，对Python系统来说，这将确保不会与用户自定义的名称冲突。通常，你将会覆写这些方法，并在里面实现你所需要的功能，以便Python调用它们。例如，当定义一个类时，你经常会覆写“\_\_init\_\_”方法。

虽然你也可以编写自己的特殊方法名，但不要这样做。

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> class C(object):  ... def \_\_mine\_\_(self):  ... pass  ...  >>> dir(C)  ... [..., '\_\_mine\_\_', ...] |

其实，很容易摆脱这种类型的命名，而只让Python内部定义的特殊名称遵循这种约定