# **Python 中的下划线，双下划线**

在 Python 中变量和方法里经常见到单下划线和双下划线，比如常见 \_\_future\_\_, \_\_all\_\_, \_\_version\_\_, \_\_author\_\_。

**from** \_\_future\_\_ **import** print\_function

**import** os

\_\_all\_\_ **=** ['function1', 'function2']\_\_version\_\_ **=** '0.01'\_\_author\_\_ **=** 'towardsdeeplearning.com'

print("Hello World")

再细分的话就是下面几种情况了，我在这里简单列举下：

1. 前置的单下划线：\_var
2. 后置的单下划线：var\_
3. 前置的双下划线：\_\_var
4. 前后置的双下划线：\_\_var\_\_
5. 单独的下划线：\_

## **1. Single Leading Underscore: “\_var”**

下划线前缀一般约定是为了提示其他程序员，以单个下划线开头的变量或方法供内部使用。PEP 8 中定义了此约定，这是最常用的 Python 编程规范。当然，这个只是一个指示性，并不是强制，Python 在“私有”和“公共”变量之间没有像 Java 明确。

class Test:

def \_\_init\_\_(self):

self.foo = 11

self.\_bar = 23

>>> t = Test()

>>> t.foo

11

>>> t.\_bar

23

这里即使加上了下划线的前缀，我们也能访问变量，但要注意的是，下划线的前缀会影响从模块导入名称的方式。

# my\_module.py:

def external\_func():

return 23

def \_internal\_func():

return 42

我们倒入的时候能看到一个奇怪的现象：

>>> from my\_module import \*

>>> external\_func()

23

>>> \_internal\_func()

NameError: "name '\_internal\_func' is not defined"

这主要是因为通过通配符导从模块中导入所有函数，Python 将不会导入带下划线的函数（或者你显式的在\_\_all\_\_ 列表里定义），应避免使用通配符导入。但是常规导入不受前缀下划线命名约定的影响：

>>> import my\_module

>>> my\_module.external\_func()

23

>>> my\_module.\_internal\_func()

42

## **2.Single Trailing Underscore: “var\_”**

当一个属性名恰好跟 Python 的关键字重名，为了直观，可以在属性名后加个\_

>>> def make\_object(name, class):

SyntaxError: "invalid syntax"

>>> def make\_object(name, class\_):

... pass

## **3.双下划线前缀：“\_\_var”**

此时 Python 解释器会重写属性名称垃圾避免子类中命名冲突，这就是所谓的 *mangling，*名字修饰，解释器以某种方式更改变量的名称，以使以后扩展类时更难产生冲突。

class Test:

def \_\_init\_\_(self):

self.foo = 11

self.\_bar = 23

self.\_\_baz = 42

使用 Python 内置的函数 dir() 查看下 Test 类的属性。

>>> t = Test()

>>> dir(t)

['\_Test\_\_baz', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_','\_\_dir\_\_',

'\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_',

'\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_',

'\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_',

'\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_bar', 'foo']

**dir()** 函数不带参数时，返回当前范围内的变量、方法和定义的类型列表；带参数时，返回参数的属性、方法列表。如果参数包含方法\_\_dir\_\_()，该方法将被调用。如果参数不包含\_\_dir\_\_()，该方法将最大限度地收集参数信息。上边返回类里变量 foo, \_bar, 但是没有发现 \_\_baz 这个变量。

强行访问的话就会报错

>>> t = Test()

>>> t.\_\_baz

AttributeError: "'Test' object has no attribute '\_\_baz'"

我们仔细看下上边的列表，发现有个叫 \_Test\_\_baz 属性，这就是所谓的*mangling ，*这样做是为了防止变量在子类中被覆盖。

我们来建立个类 ExtendedTest，继承自 Test，试图在构造函数里覆盖已经存在的属性。

class ExtendedTest(Test):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.foo = 'overridden'

self.\_bar = 'overridden'

self.\_\_baz = 'overridden'

来看下 ExtendedTest 实例中 foo, \_bar, 和 \_\_baz 变量。

>>> t2 = ExtendedTest()

>>> t2.foo

'overridden'

>>> t2.\_bar 'overridden'

>>> t2.\_\_baz

AttributeError: "'ExtendedTest' object has no attribute '\_\_baz'"

当试图访问 t2.\_\_baz 时，又出现了 AttributeError，这其实就是 *mangling，查看下属性确实没有*\_\_baz *这个属性。*

>>> dir(t2)

['\_ExtendedTest\_\_baz', '\_Test\_\_baz', '\_\_class\_\_', '\_\_delattr\_\_',

'\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_',

'\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_',

'\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_',

'\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_',

'\_bar', 'foo', 'get\_vars']

\_\_baz 变成了 \_ExtendedTest\_\_baz，基类 Test 的变量变成了 \_Test\_\_baz，我们试图访问下：

>>> t2.\_ExtendedTest\_\_baz 'overridden'

>>> t2.\_Test\_\_baz

42

那么对于方法来说，mangling 同样适用。

class MangledMethod:

def \_\_method(self):

return 42

def call\_it(self):

return self.\_\_method()

>>> MangledMethod().\_\_method()

AttributeError: "'MangledMethod' object has no attribute '\_\_method'"

>>> MangledMethod().call\_it()

42

## **4. 前后双下划线**

常用于 \_\_init\_\_, \_\_call\_\_,\_\_iter\_\_, \_\_next\_\_这些方法里，但是，通常我们自己的方法名最好不要用。

class PrefixPostfixTest:

def \_\_init\_\_(self):

self.\_\_bam\_\_ = 42

>>> PrefixPostfixTest().\_\_bam\_\_

42

## **5. Single Underscore: “\_”**

有时候函数返回值不止一个，但有些变量我们不需要，就可以使用 \_ 来当个用不到的变量。

>>> for \_ in range(32):

... print('Hello, World.')