Python必备核心技能 - 模块module（内容超详细，举例说明很容易懂）

Python**模块和包**是和变量，函数，类同等重要的基础的基础。因为它们是Python组织程序的方式。掌握不好，你很难学习和理解新的知识，会处处碰壁。是掌握Python必备的核心技能。

> 在硬核Python私教班课程中，我把所有关键技能总结为**Python 36技**，模块和包是其中之一。如果你想快速系统的学习Python，建立办公自动化，数据分析等实战技能，需要有专业老师指导你解决问题，可以考虑麦叔私教课，物美价廉。有兴趣咨询maishu1024。

本文详细讲解了：

- 模块和概念；
- 创建自己的模块；
- Python模块的查找路径；
- import语句的各种用法；
- 创建模块和核心技巧；

模块化编程是指将编程任务分解为单独的，较小的，更易于管理的子任务或模块的过程。然后可以像构建模块一样将各个模块拼凑在一起以创建更大的应用程序。

程序模块化代码有几个优点：

- **简单性**：模块通常只关注问题的一个相对较小的部分，而不是关注整个问题。如果你只关注单个模块，那么只有一个更小的问题域需要处理。这使得开发更容易，更不容易出错。
- **可维护性**：分成多个小文件，修改的时候只要修改相关的文件就行了。而且可以多人协作，同时修改而不会产生冲突。
- **可重用性**：单个模块中定义的功能可以被应用程序的其他部分轻松地重用（通过适当定义的接口）。这减少了重复代码。
- **作用域**：模块通常会定义一个单独的命名空间，这有助于避免程序不同区域中的名字发生冲突。（Python禅的信条之一是：命名空间是一个很棒的发明）

函数，模块和包都是Python中促进代码模块化的机制。

Python模块：概述

在Python中定义模块的方式有三种：

- 用Python编写的模块，我们写的主要是这一类模块

- 用C语言编写模块，并在运行时动态加载模块，例如re（正则表达式）模块。这种模块看不到源代码。
- 包含在解释器中的内置模块，比如itertools模块。

在这三种情况下，访问模块内容的方式都是相同的: 使用import语句。

本文重点主要放在用Python编写的模块上。用Python编写的模块很酷的一点是，它很容易。你所需要做的就是创建一个包含合法Python代码的文件，然后给该文件一个后缀名为.py的名称。就是这样!不需要特殊的语法或魔法。你就创建了一个模块。

例如，假设你创建了一个名为mod.py的文件，其中包含以下内容:

```
s = "这是我的第一个模块，我是麦叔，喜欢请关注我."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass
```

mod.py中定义了几个对象:

- s (一个字符串)
- a (一个列表)
- foo() (一个函数)
- Foo (一个类)

假设mod.py保存在你电脑上的某一个位置，这些对象可以通过如下方式导入模块来访问:

```
>>> import mod
>>> print(mod.s)
这是我的第一个模块，我是麦叔，喜欢请关注我.
>>> mod.a
[100, 200, 300]
>>> mod.foo(['你好', '我好', '大家好'])
arg = ['你好', '我好', '大家好']
>>> x = mod.Foo()
```

```
>>> x
<mod.Foo object at 0x03C181F0>
```

模块搜索路径

继续上面的例子，让我们看看当Python执行语句时会发生什么:

```
import mod
```

当解释器执行上面的import语句时，它会依次去下面的目录搜索mod.py，搜索不到就报错:

- 运行代码的目录，如果使用交互式Python，则为当前目录
- 环境变量`PYTHONPATH`中的目录。
- 安装Python时配置的与安装相关的目录列表

可以通过sys.path来得到所有搜索路径。

```
>>> import sys
>>> sys.path
['', 'C:\\Users\\maishu\\Documents\\Python\\doc', 'C:\\Python36\\Lib\\idlelib',
'C:\\Python36\\python36.zip', 'C:\\Python36\\DLLs', 'C:\\Python36\\lib',
'C:\\Python36', 'C:\\Python36\\lib\\site-packages']
```

> 注意:sys.path 和你的python环境有关。上面的代码在你的电脑上看起来会有些许不同。

因此，要确保找到你的模块，你需要执行以下操作之一:

- 如果是交互环境，把mod.py保存到执行命令的当前目录
- 将mod.py所在目录放到PYTHONPATH环境变量中
  - 或者:把mod.py文件放到已经存在PYTHONPATH中的某一个目录
- 把mod.py放在python安装相关的目录中

实际上还有一个额外的选项:你可以将模块文件放在你选择的任何目录中，然后临时修改sys.path，以便包含该目录。例如，在这种情况下，你可以把mod.py放在目录C:\Users\maishu中，然后使用以下代码:

```
>>> sys.path.append(r'C:\Users\maishu')
>>> sys.path
['', 'C:\\Users\\maishu\\Documents\\Python\\doc', 'C:\\Python36\\Lib\\idlelib',
'C:\\Python36\\python36.zip', 'C:\\Python36\\DLLs', 'C:\\Python36\\lib',
'C:\\Python36', 'C:\\Python36\\lib\\site-packages', 'C:\\Users\\maishu']
>>> import mod
```

一旦模块被导入，你就可以通过模块的 `__file__` 属性来确定它被找到的位置:

```
>>> import mod
>>> mod.__file__
'C:\\Users\\maishu\\mod.py'

>>> import re
>>> re.__file__
'C:\\Python36\\lib\\re.py'
```

import语句

最简单的形式是上面所示的:

```
import <module_name>
```

请注意，这种简单的形式只是引用了模块本身，模块内的东西并没有直接引入，不能直接用名字访问，而要通过**模块.变量**的形式。

模块会创建一个单独的名称空间，防止和别的模块发生冲突。这样就能以模块名作为前缀，通过**点**访问模块内的东西，如下所示。

在下列import语句之后，mod被引入到本文件中。因此，可以引用mod:

```
```

```
>>> import mod
>>> mod
<module 'mod' from 'C:\\Users\\maishu\\Documents\\Python\\doc\\mod.py'>
```

但是s和foo仍然在属于mod内部的变量和函数，不能直接使用:

```
>>> s
NameError: name 's' is not defined
>>> foo('quux')
NameError: name 'foo' is not defined
```

必须加上mod前缀才可以:

```
>>> mod.s
'If Comrade Napoleon says it, it must be right.'
>>> mod.foo('quux')
arg = quux
```

几个用逗号分隔的模块可以在一个import语句中指定:

```
import <module_name>[, <module_name> ...]
```

from <module_name> import <name(s)>

import语句的另一种形式允许将模块中的单个对象直接导入:

```
from <module_name> import <name(s)>
```

执行上面的语句后， 可以在不使用前缀的调用者环境中被引用:

```
```

```
>>> from mod import s, foo
>>> s
'这是我的第一个模块，我是麦叔，喜欢请关注我.'
>>> foo('你好')
arg = 你好

>>> from mod import Foo
>>> x = Foo()
>>> x
<mod.Foo object at 0x02E3AD50>
```

因为这种形式的导入将对象名称直接引入当前文件，任何已经存在的具有相同名称的对象都将被覆盖:

```
>>> a = ['foo', 'bar', 'baz']
>>> a
['foo', 'bar', 'baz']

>>> from mod import a
>>> a
[100, 200, 300]
```

甚至可以一下子不加区别地从一个模块中导入所有内容:

```
from <module_name> import *
```

这将把模块中的所有对象的名称引入当前文件，但不包括以下划线(_)开头的内容。

例如:

```
>>> from mod import *
>>> s
'这是我的第一个模块，我是麦叔，喜欢请关注我.'
>>> a
[100, 200, 300]
```

```
>>> foo
<function foo at 0x03B449C0>
>>> Foo
<class 'mod.Foo'>
```

在大规模的项目中并不推荐这样做。这有一点危险，因为这会在引入全部名称。除非你对它们都很了解，并且确信不会有冲突，否则你很有可能会无意中覆盖现有的变量。然而，当你只是为了测试或学习目的而使用交互式解释器时，此语法非常方便，因为它可以让你快速访问模块所提供的所有内容，而无需大量输入。

from <module_name> import as <alt_name>

也可以导入单独的对象，并且给它设置一个别名:

```
from <module_name> import <name> as <alt_name>[, <name> as <alt_name> …]
```

这使得可以直接将名称引入，但避免与之前存在的名称冲突:

```
>>> s = 'foo'
>>> a = ['foo', 'bar', 'baz']

>>> from mod import s as string, a as alist
>>> s
'foo'
>>> string
'这是我的第一个模块，我是麦叔，喜欢请关注我.'
>>> a
['foo', 'bar', 'baz']
>>> alist
[100, 200, 300]
```

import <module_name> as <alt_name>

你也可以用这种方式导入整个模块:

```
import <module_name> as <alt_name>
```

```
>>> import mod as my_module
>>> my_module.a
[100, 200, 300]
>>> my_module.foo('qux')
arg = qux
```

模块内容也可以从函数中导入。在这种情况下，模块只有在函数被调用才会被导入:

```
>>> def bar():
...     from mod import foo
...     foo('corge')
...

>>> bar()
arg = corge
```

然而，Python3不允许从函数内部使用import *:

```
>>> def bar():
...     from mod import *
...
SyntaxError: import * only allowed at module level
```

最后，带有except ImportError子句的try语句可以用来防止失败的导入尝试:

```
>>> try:
...     # Non-existent module
...     import baz
... except ImportError:
...     print('Module not found')
...
```

```
Module not found
```

```
>>> try:
...     # Existing module, but non-existent object
...     from mod import baz
... except ImportError:
...     print('Object not found in module')
...

Object not found in module
```

dir()函数

内置函数dir()返回一个模块中定义的所有名称列表（函数，全局变量等）。如果没有参数，它会在当前局部符号表中生成一个按字母排序的名称列表:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> qux = [1, 2, 3, 4, 5]
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'qux']

>>> class Bar():
...     pass
...
>>> x = Bar()
>>> dir()
['Bar', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'qux', 'x']
```

请注意上面对dir()的第一次调用是如何列出几个已经在命名空间中的名称的。当定义了新的名称(qux, Bar, x)时，它们会出现在后续的dir()调用中。

这对于确定import语句到底给命名空间添加了什么是很有用的:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> import mod
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'mod']
>>> mod.s
'If Comrade Napoleon says it, it must be right.'
>>> mod.foo([1, 2, 3])
arg = [1, 2, 3]

>>> from mod import a, Foo
>>> dir()
['Foo', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'a', 'mod']
>>> a
[100, 200, 300]
>>> x = Foo()
>>> x
<mod.Foo object at 0x002EAD50>

>>> from mod import s as string
>>> dir()
['Foo', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'a', 'mod', 'string', 'x']
>>> string
'If Comrade Napoleon says it, it must be right.'
```

当给dir()一个模块名的参数时，dir()列出在该模块中定义的名称:

```
>>> import mod
>>> dir(mod)
```

```
['Foo', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'a', 'foo', 's']
```

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
>>> from mod import *
>>> dir()
['Foo', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'a', 'foo', 's']
```

将模块作为脚本执行

任何包含模块的.py文件本质上也是一个Python脚本，没有任何理由不能像Python脚本那样执行它。

下面是mod.py的定义:

mod.py

```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass
```

这可以作为一个脚本运行:

```
C:\Users\maishu\Documents>python mod.py
C:\Users\maishu\Documents>
```

没有错误，所以它显然是有效的。当然，这不是很有趣。就像它写的那样，它只定义对象。它不会对它们做任何事情，也不会生成任何输出。

让我们修改一下上面的Python模块，这样当它作为脚本运行时就会生成一些输出：

mod.py

```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass

print(s)
print(a)
foo('quux')
x = Foo()
print(x)
```

现在应该更有趣了：

```
C:\Users\maishu\Documents>python mod.py
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<__main__.Foo object at 0x02F101D0>
```

不幸的是，现在它也会在作为模块导入时生成输出：

```
>>> import mod
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<mod.Foo object at 0x0169AD50>
```

这可能不是你想要的。模块在导入时通常不会生成输出。

如果能够区分文件是作为模块加载的，还是作为独立脚本运行的，不是很好吗？

这当然是可以做到的！

当.py文件作为模块导入时，Python将特殊的变量 `__name__` 设置为模块的名称。但是，如果一个文件作为一个独立的脚本运行， `__name__` 被设置为字符串 `'__main__'` 。基于这个，你可以辨别哪种情况是在运行时发生的，并相应地改变行为：

mod.py

```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass

if (__name__ == '__main__'):
    print('Executing as standalone script')
    print(s)
    print(a)
    foo('quux')
    x = Foo()
    print(x)
```

现在，如果你作为一个脚本运行，你会得到输出：

```
C:\Users\maishu\Documents>python mod.py
Executing as standalone script
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<__main__.Foo object at 0x03450690>
```

但如果你以模块的形式导入，不会有输出：

```
>>> import mod
>>> mod.foo('grault')
arg = grault
```

为了测试模块中包含的功能，模块通常被设计为能够作为独立的脚本运行。这被称为单元测试。例如，假设你已经创建了一个包含阶乘函数的模块fact.py，如下所示:

fact.py

```
def fact(n):
    return 1 if n == 1 else n * fact(n-1)

if (__name__ == '__main__'):
    import sys
    if len(sys.argv) > 1:
        print(fact(int(sys.argv[1])))
```

可以将该文件视为一个模块，并导入fact()函数:

```
>>> from fact import fact
>>> fact(6)
720
```

但它也可以通过在命令行中传递一个整数参数来独立运行，以进行测试:

```
C:\Users\maishu\Documents>python fact.py 6
720
```

重新加载模块

为了提高效率，一个模块在每次解释器会话中只加载一次。这对于函数和类定义来说很好，它们通常构成模块内容的大部分。但是模块也可以包含可执行语句，通常用于初始化。请注意，这些语句只会在第一次导入模块时执行。

考虑下面的mod.py文件:

mod.py

```
```

```
a = [100, 200, 300]
print('a =', a)
```

```
>>> import mod
a = [100, 200, 300]
>>> import mod
>>> import mod

>>> mod.a
[100, 200, 300]
```

print()语句不会在后续导入中执行。

如果你对模块做了更改并需要重新加载它，你需要重新启动解释器或使用模块importlib中的重载函数: reload()

```
>>> import mod
a = [100, 200, 300]

>>> import mod

>>> import importlib
>>> importlib.reload(mod)
a = [100, 200, 300]
<module 'mod' from 'C:\\Users\\maishu\\Documents\\Python\\doc\\mod.py'>
```

回顾

这个教程本来包含模块和包两个部分，但是由于篇幅限制，本文只包含了模块（绿色部分）。后续内容会一周内更新。请关注本公众号：麦叔编程。

在公众号回复**模块**或者**包**，可以下载本文的PDF版本，获得后续文章的更新。

教程目录：

- 如何创建Python模块
- Python解释器搜索模块的位置

- import的各种用法举例详解
- 模块最常用的技巧：可独立执行的模块
- 如何将模块组织成包和子包【下一节】
- 如何控制包初始化【下一节】

如果你想了解更多，请关注公众号**麦叔编程**。