第6章. 模块与作用域

6-1. 模块基础

（教程：6.1～6.3）
（语言参考手册：3.2、7.11）
（标准库：内置类型、types）

第4章讨论的函数和第5章讨论的类是Python程序的基本构建单元，原则上足以构建任意复杂的程序，但前提是将整个程序写在一个文件内。显然，现代程序的规模使这种做法不切实际，而且模块化思想也暗示了将构建单元分散在多个文件中——这样更便于管理、维护和升级程序，因为一个文件的损坏不会影响到其他文件。由于在Python中一切皆对象，而文件显然不能用一个类或一个函数来表示，所以需要引入一种新的构建单元——“模块”。类能够以属性的形式包含函数，而一个文件可以包含多条类定义语句和函数定义语句，所以这三种构建单元按照粒度从细到粗依次为：函数 → 类 → 模块。

标准库中的types模块定义了ModuleType来代表模块类型，这在第3章已经提到了。这里要强调ModuleType可以直接被实例化，其语法为：

```
class types.ModuleType(name, doc=None)
```

其中name参数用于设置模块名，doc参数用于设置模块的文档。这使我们可以手工创建一个模块对象并研究其具有哪些属性。请执行下面的命令行和语句：

```
$ python3

>>> import types
>>> m = types.ModuleType('mymodule', 'Document for mymodule.')
>>> m.__dict__
{'__name__': 'mymodule', '__doc__': 'Document for mymodule.',
'__package__': None, '__loader__': None, '__spec__': None}
>>> m.__name__
'mymodule'
>>> m.__doc__
'Document for mymodule.'
>>> m.__annotations__
{}
>>> m.__file__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'mymodule' has no attribute '__file__'. Did you
mean: '__name__'?
>>> m.__path__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module '__main__' has no attribute '__path__'. Did you
mean: '__name__'?
>>> m.__cached__
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
AttributeError: module 'mymodule' has no attribute '__cached__'
>>>
```

注意表3-3列出了模块的6个特殊属性，而上面例子中模块对象m仅具有前3个，这是因为它是手工创建的，未关联到任何文件。此外，手工创建的模块对象的__annotations__特殊属性引用的是None。模块对象还有3个实例属性：

➤ __package__：引用一个字符串，为该模块所属包的名称。如果该模块不属于任何包，则该属性引用None。
➤ __loader__：引用加载该模块的加载器。如果使用默认加载器，则该属性引用None。
➤ __spec__：引用该模块的导入系统相关状态的记录。如果该模块是被导入的，则该属性将引用一个importlib.machinery.ModuleSpec类型的对象。

注意这些属性的属性名都符合“__*__”的模式，但因为是实例属性而非类属性，所以并不属于魔术属性。

虽然我们可以手工创建模块对象，但这样的模块对象没有什么用。现实中模块对象按照来源可以分为三类（用C编写的扩展模块其实也是模块对象的一个来源，但它们在逻辑上等同于模块）：

➤ 内置模块：源自Python解释器自身（标准库提供了访问这些模块的接口），用C语言编写。
➤ 模块：源自标准库中的Python脚本（后缀名.py），用户自己编写的Python脚本（后缀名.py），以及这些Python脚本翻译成的字节码文件（后缀名.pyc）。
➤ 冻结模块：源自冻结包袱（会在第17章详细讨论）。

代表模块或冻结模块的模块对象具有__file__特殊属性，引用的字符串为相应文件在文件系统中的路径。

Python解释器成功启动后，部分内置模块对应的模块对象就已经存在了，以使Python语句可以被解释执行，并使我们可以使用内置函数、内置常量、内置类型和内置异常。此外，Python解释器会自动创建一个名为“__main__”的模块对象，即所谓的“主模块（main module）”：如果在启动Python解释器时指定了Python脚本，则主模块代表该脚本；否则，主模块相当于执行“types.ModuleType('__main__')”得到的模块对象。其他内置模块和模块都是直接或间接地由主模块通过import语句（以及等价方法，下同）导入的，在导入时会自动创建相应的模块对象。

而对于冻结包袱来说，当它被执行时在逻辑上也会按照上述顺序创建所有模块对象，只不过这些模块对象都源自冻结包袱本身，包括对应Python解释器的模块。这些模块就仿佛被“冻结”在冻结包袱中，因此而得名。为了便于讨论，下面暂时不考虑冻结模块。

与函数对象和类对象不同，模块对象是不可调用的，有意义的是它们的实例属性。事实上，我们可以把模块对象想象成容器，其来源代码中定义的标识符最终会成为它们的实例属性。在仅考虑用Python实现的模块时，这些标识符可以是：

- 通过赋值语句定义的变量。
- 通过函数定义语句定义的函数。
- 通过类定义语句定义的类。

而用C实现的模块依然定义了这三种标识符，只不过实现方式不同。（C中没有“类”的概念，因此定义Python中的类要采用特殊手段。）

前面已经提到了，Python脚本不会直接创建模块对象，而是通过import语句导入模块到当前作用域（会在下一节解释）。

import语句有三种语法。不论用import语句的那种语法，当导入的标识符已经存在时，都相当于对该标识符进行了一次赋值。

---

import语句的第一种语法为：

```
import module [as identifier], ...
```

其中module为目标模块的模块名；identifier用于对模块改名，仅当模块名与当前作用域中的其他标识符冲突时才需要使用。上述语法中的逗号和省略号表示允许通过一条import语句导入多个模块，但这等同于通过多条import语句导入这些模块（每条语句仅导入一个模块）。下面仅考虑一次导入一个模块的情况，此时将按照如下步骤执行：

- 步骤一：查找module参数指定的模块，如果相应的模块对象尚未被创建则创建它，即完成指定模块的加载。如果找不到指定模块，则抛出ModuleNotFoundError异常。
- 步骤二：在当前作用域中定义一个标识符，使它引用对应指定模块的模块对象，进而使我们能够以“module.attribute”的形式访问由指定模块定义的标识符。

前面几章中的例子已经多次使用了import语句的第一种语法从标准库导入所需模块，例如cache.py和lru_cache.py。下面的例子则用import语句的第一种语法导入我们自己用Python编写的模块。首先请将如下代码保存到module1.py：

```
#定义一个变量。
a = 1

#定义一个私有变量。
_b = 100
```

```
#定义一个函数。
def f1():
    return 'f1'


#定义一个私有函数。
def _f2():
    return 'f2'


#定义一个类。
class C1():
    def echo(self):
        return 'C1'


#定义一个私有类。    注意这里其实违背了PEP 8的相应推荐。
class _C2():
    def echo(self):
        return 'C2'
```

然后将module1.py放在工作目录，执行如下命令行和语句：

```
$ python3

>>> import module1
>>> module1.a
1
>>> module1._b
100
>>> module1.f1()
'f1'
>>> module1._f2()
'f2'
>>> c1 = module1.C1()
>>> c1.echo()
'C1'
>>> c2 = module1._C2()
>>> c2.echo()
'C2'
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'c1', 'c2', 'module1']
>>>
```

从上述语句的执行结果可以看出如下两点：

➤ 对于用Python编写的模块来说，模块名默认为相应文件去掉.py后缀名后剩下的部分。
➤ 以import语句的第一种语法导入一个模块后，可以通过相应模块对象访问到该模块定义的所有标识符，不论它是否以“_”开头。

此外，该例子用到了这样一个技巧：以省略了object参数的方式调用第3章中介绍过的内置函数dir()。这会显示主模块当前的实例属性，其中包含了所有被导入的标识符，本例子中是“module1”。

下面我们做一个实验。在工作目录下创建“MyModules”目录，将module1.py移动到该目录下，然后再执行如下命令行和语句：

```
$ python3

>>> import module1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'module1'
>>>
```

结果显示找不到module1模块，导入失败。这是因为Python解释器是按照如下步骤搜索用户自己编写的模块的：

➤ 步骤一：在工作目录下搜索。（注意如果在启动Python解释器时指定了Python脚本，则工作目录会被切换到该Python脚本所在目录。）
➤ 步骤二：在环境变量PYTHONPATH取值包含的目录下搜索。（PYTHONPATH取值一个以“:”分隔的字符串序列，其中的每一项都是一个目录的文件系统路径。默认情况下它的取值是一个空字符串。）
➤ 步骤三：在Python解释器内置的模块目录列表包含的目录中搜索。

由于PYTHONPATH的取值和Python解释器内置的模块目录列表默认都不包含MyModules目录，所以对module1的搜索会失败。

解决上述问题的第一种办法是修改环境变量PYTHONPATH。如果你在使用Unix或类Unix操作系统，那么请通过如下命令行将MyModules目录添加到PYTHONPATH中去：

```
$ export PYTHONPATH="$PYTHONPATH:./MyModules"
```

这意味着此后Python解释器会在工作目录下的MyModules子目录中搜索模块。请通过如下命令行和语句验证：

```
$ python3

>>> import module1
>>>
```

然而这种办法的缺点是：

- 该设置仅对当前shell有效，重新打开终端后就会失效。
- 该设置无法固化在导入模块的Python脚本中。

解决上述问题的第二种办法是将MyModules目录添加到Python解释器内置的模块目录列表中去，而这就需要使用标准库中的sys模块提供的path属性。sys是与运行时服务相关的模块之一，将在第7章被详细讨论。这里只需要知道sys.path引用的列表对象就是储存内置模块目录列表的地方，可以通过该对象的append属性添加目录或zip文件。请重新打开终端（以使对PYTHONPATH的设置失效）：

```
$ python3
```

然后执行如下语句：

```
>>> import sys
>>> sys.path.append("./MyModules")
>>> import module1
>>>
```

由于该办法只用到了Python语句，不需要使用shell命令，因此可以固化到Python脚本中，使得该Python脚本所在目录的子目录MyModules中的模块都能够被找到。

最后，module1被成功导入后，module1.py所在目录下会自动生成“__pycache__”子目录，并包含文件“module1.cpython-311.pyc”。以.pyc为后缀名的文件储存的是字节码，即Python脚本解释执行的中间产物。

当其他Python脚本导入module1时，Python解释器先找到module1.py文件，然后在它所在目录下的__pycache__子目录中查找相应的.pyc文件，如果找到则将它的创建日期与module1.py进行比较，如果前者晚于后者则直接使用.pyc文件。这一机制是为了加快模块的加载速度，但正如第1章已经说明的，该机制本质上是跳过了将Python代码翻译成IL的过程，而该过程本身就比较快，所以性能提升不会太大。

有了.pyc文件后，也可以移除.py文件，直接以.pyc文件来实现模块，这被称为“无源文件发行版”。需要做的只是将“module1.cpython-311.pyc”重命名为“module1.pyc”，然后将其放在module1.py所在目录下，移走module1.py即可。注意，当存在同名的.py文件和.pyc文件时，Python解释器会对.py文件进行重新解释，忽略.pyc文件。

至此我们已经对用import语句的第一种语法导入我们自己编写的模块有了足够的了解。下面讨论import语句的第二种语法：

```
from relative_module import identifier [as identifier], ...
```

它也可以等价的写为（也就是给标识符列表添加括号）：

```
from relative_module import (identifier [as identifier], ...)
```

其中relative_module也是模块名，只不过在包存在的情况下允许使用相对模块名（包会在后面讨论）。import语句的这种语法执行时在步骤一与第一种语法完全相同，但步骤二变成了：在当前作用域中定义通过as子句指定的标识符，使它们引用relative_module模块中指定标识符引用的对象。（如果省略了as子句，则两个标识符同名。）这使我们可以直接以“identifer”的形式访问由指定模块定义的标识符。此外，这种导入方式还可以精确控制导入哪些标识符。

同样，在前面几章的例子中已经多次使用了import语句的第二种语法从标准库中的模块导入所需标识符，例如BranchAttr.py和clock_descriptor.py。下面让我们用import语句的第二种语法从module1中导入标识符。请执行如下命令行和语句：

```
$ python3

>>> from module1 import _b as b, _f2 as f2, _C2 as C2
>>> b
100
>>> f2()
'f2'
>>> c2 = C2()
>>> c2.echo()
'C2'
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> f1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'f1' is not defined. Did you mean: 'f2'?
>>> c1 = C1()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'C1' is not defined. Did you mean: 'C2'?
>>> dir()
['C2', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'b', 'f2']
>>>
```

可以看出，我们从module1中选择性地导入了_b、_f2和_C2，并将它们分别重命名为b、f2和C2。

import语句的第三种语法其实是第二种语法的缩略形式：

```
from relative_module import *
```

其中“*”表示指定模块中的“所有可导入的标识符”，默认情况下为那些不以下划线开头的标识符。请通过如下命令行和语句验证：

```
$ python3

>>> from module1 import *
>>> a
1
>>> _b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_b' is not defined. Did you mean: '_'?
>>> f1()
'f1'
>>> _f2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_f2' is not defined
>>> c1 = C1()
>>> c1.echo()
'C1'
>>> c2 = C2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'C2' is not defined. Did you mean: 'C1'?
>>> dir()
['C1', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'a', 'f1']
>>>
```

这里我们从module1中选择性地导入了a、f1和C1，而_b、_f2和_C2被跳过，这也是模块中以下划线开头的标识符被视为私有的原因。

需要强调的是，我们可以通过在Python脚本中定义特殊标识符__all__来指定哪些标识符在使用import语句的第三种语法时被导入。

__all__必须引用一个字符串列表，每个字符串对应一个可导入的标识符。下面修改module1.py为module2.py，即在顶部添加如下语句：

```
#定义可导入变量列表。
__all__ = ['a', '_f2', 'C1', '_C2']
```

请重新启动Python解释器，然后通过如下语句验证：

```
$ python3

>>> from module2 import *
>>> a
1
>>> _b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_b' is not defined. Did you mean: '_'?
>>> f1()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'f1' is not defined. Did you mean: 'C1'?
>>> _f2()
'f2'
>>> c1 = C1()
>>> c1.echo()
'C1'
>>> c2 = _C2()
>>> c2.echo()
'C2'
>>> dir()
['C1', '_C2', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', '_f2', 'a']
>>>
```

该结果说明被选择性地导入的标识符是a、_f2、C1和_C2。PEP 8建议在每个模块中都通过定义__all__来明确可导入的标识符，亦即该模块对外界提供的接口。

最后，在第2章提到过，启动Python解释器时可以通过-m指定执行一个模块。当该模块通过Python脚本实现时，本质上与直接执行该Python脚本相同。

这里需要说明的是，与通过import语句导入一个模块不同，通过-m执行一个模块会创建主模块。这意味着虽然仍会创建对应指定模块的模块对象，但该模块对象的__name__特殊属性会引用“__main__”，以表明创建的是主模块。

这给我们提供了一种判断Python脚本是否被作为主模块执行的方法，并可以据此设置仅当脚本作为主模块时才执行的代码，以及仅当脚本没有作为主模块时才执行的代码。请看下面的例子：

```
if __name__ == '__main__':
    #仅当作为主模块时才执行的代码。
    print("I am executed directly!")
else:
    #仅当没有作为主模块时才执行的代码。
    print("I am imported.")
```

请将上述代码保存为module3.py，然后通过如下命令行和语句验证：

```
$ python3 module3.py
I am executed directly!

$ python3 -m module3
I am executed directly!

$ python3
>>> import module3
I am imported.
>>>
```

6-2. 名字空间和作用域

（教程：4.7、9.2）
（语言参考手册：3.2、4.1、4.2、7.2、7.5、7.12、7.13）
（标准库：内置函数）

至此本书介绍完了Python程序的所有构建单元，接下来讨论对所有编程语言都很重要的一个概念——“作用域（scopes）”。考虑这样的问题：大型程序通常需要成千上万人参与，通过人工协调来保证他们编写的代码中没有冲突的标识符将带来极高的管理成本。而该问题的解决方案是让每个标识符都有自己的作用域，利用作用域降低标识符发生冲突的概率，再用一套明确的作用域规则来规定标识符发生冲突时如何解决。这可以极大降低人工协调的难度，有效节省管理成本。

作用域是个静态概念，是在Python脚本被解释器进行第一轮处理时就确定的，如果存在标识符冲突则会报错。然而作用域的确定规则却建立在“名字空间（namespaces）”上，而名字空间是在Python脚本被执行时动态生成的。

作用域和名字空间的这种微妙关系使得理解作用域规则相对困难，对于任何编程语言来说都是其语法中的难点。

这里首先阐明一点：经过本书前面部分的讨论，相信你已经发现，Python中的常量是用变量实现的，而变量或者是函数的属性，或者是类的属性，或者是模块的属性；函数或者是类的属性，或者是模块的属性；类是模块的属性。现在想象一个代表整个世界的虚拟对象，所有模块都是它的属性，那么Python中的所有标识符都将是属性名。第3章已经说明了，属性和变量在逻辑上是等价的。所以可认为Python中的每个标识符都是一个变量名，而在讨论作用域时，习惯上使用“变量”这个术语，而非“标识符”这个术语。

下面阐述名字空间这一概念。第3章已经说明，每个对象都具有若干特殊属性（至少包括__class__），都有一个变量字典和/或一个C数组来储存其实例属性，且都通过__class__引用的类的继承链来访问其类属性。一个对象的特殊属性、实例属性和类属性的全体，就构成了该对象的名字空间。反过来，一个对象的名字空间决定了可以通过该对象访问到哪些标识

符。对象被创建之后，它的名字空间就保证了与其他对象的名字空间不会发生冲突（虽然同类对象的名字空间可以有交集）。

因此，我们的关注点变成了Python解释器在进行第一轮处理时，如何确定每个标识符都引用了哪个对象。注意对于“identifier1.identifier2”的形式给出的两个标识符，只需要确定identifier1引用的对象，就可以确定identifier2所属名字空间，因此难点仅在于那些没有明确指定其所属名字空间的标识符。

到目前为止，我们已经介绍了第3章中列出的导致标识符被绑定到对象的7个操作中的前6个，即赋值语句、赋值表达式、函数定义、类定义、模块的导入、函数被调用时的参数传递。然而除了最后一个绑定操作的有效范围总是函数体，前5个绑定操作的有效范围因它们出现的位置不同而不同。

➤ 首先，如果绑定操作出现在任何函数体和类体之外，则通过它们定义的标识符将成为对应该Python脚本的模块对象的实例属性，该操作的有效范围是从标识符被定义的位置开始直到脚本末尾。

我们称以这种方式定义的标识符为“全局变量（global variables）”，因为它们一旦被定义，直到相应Python脚本执行完成前都可以访问（特别对于主模块来说直到程序运行完成前都可以访问），且访问它们不需要用到模块名。

我们把模块的名字空间称为“全局名字空间（global namespaces）”，其内的标识符都是全局变量。因此Python有多个全局名字空间，与大多数编程语言都不同。特别的，用于支持内置函数、内置常量、内置类型和内置异常的模块（其实就是builtins模块）的全局名字空间又被称为“内置名字空间（built-in namespace）”。

需要强调的是，定义全局变量的操作必须被执行才有效。请看下面的例子：

```
$ python3

>>> a = 1
>>> if a > 0:
...     b = 0
... else:
...     c = 0
...
>>> b
0
>>> c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
>>>
```

表面上看，a、b和c都是全局变量。然而由于执行if语句的结果只会定义b而不会定义c，所以事实上c并不是全局变量。

➤ 其次，如果绑定操作出现在某个类体内，又位于任何函数体外，则通过它定义的标识符将成为相应类对象的实例属性，因此该操作有效范围是从标识符被定义的位置开始直到类体末尾。

只有在该类体内才可以不通过类或对象直接访问该标识符本身，例如：

```
>>> class A:
...     x = 0
...     y = x
...
>>> a = A()
>>> A.y
0
>>> a.y
0
>>>
```

以这种方式定义的标识符为“本地变量（local variables）”的一种。

➤ 最后，如果绑定操作出现在某个函数体内，则通过它们定义的标识符并不会成为相应函数对象的实例属性，而是以字节码的形式固化在相应函数对象的特殊属性__code__引用的代码对象中。仅当该函数被调用时，该标识符才会在相应帧对象的f_locals属性引用的变量字典中出现。该操作的有效范围是从标识符被定义的位置开始直到函数体末尾。

以这种方式定义的标识符为本地变量的另一种。特别的，函数的形式参数的有效范围是整个函数体，因此形式参数可以看成本地变量的特例。

在明确了绑定操作的有效范围后，就可以解释什么是作用域了：一个没有明确指定其所属名字空间的标识符的作用域，是Python脚本中从定义该标识符的语句开始，直到该操作的有效范围内的最有一条语句为止。

换句话说，全局变量的作用域是从它被定义处到相应Python脚本的末尾，而本地变量的作用域是从它被定义处到相应类体或函数体的末尾。

需要注意的是，作用域可以重叠。例如类定义语句和函数定义语句中的本地变量的作用域可以被它们所在Python脚本中的全局变量的作用域完全覆盖。此外，由于函数调用会形成一个函数栈，下游（靠近栈顶）的帧对象对应的函数体会被上游（靠近栈底）的帧对象对应的函数体包含。在发生作用域重叠时，判断一个（没有明确指定其所属名字空间的）标识符属于哪个名字空间的规则可以概括为如下算法：

➤ 步骤一：从最内层作用域对应的名字空间开始搜索，如果找不到则退到外一层作用域，……，直到退到全局名字空间。
➤ 步骤二：如果在全局名字空间中依然找不到，则在内置名字空间中继续搜索。
➤ 步骤三：如果在内置名字空间中依然找不到，则抛出NameError异常。

下面的例子说明了上述规则：

```
#标识符a是全局变量。
a = -99


#标识符outer也是全局变量。
def outer():
    #标识符b是本地变量，其作用域在outer()的函数体内。
    b = 35

    #标识符inner也是本地变量，其作用域也在outer()的函数体内。
    def inner():
        #标识符c是本地变量，其作用域在inner()的函数体内。
        c = 2
        #这里使用了标识符abs、a、b和c。
        return abs(a + b*c)

    #这里使用了标识符inner。
    return inner()
```

请将上述代码保存为scope1.py，然后通过下面的命令行和语句验证：

```
$ python3 -i scope1.py

>>> outer()
29
>>>
```

该结果说明，在调用函数inner()时，可以访问到outer()中定义的本地变量b、全局变量a和内置名字空间中的abs。

从上述规则可以推知，当两个作用域的重叠部分发生标识符名字冲突时，默认会将该标识符视为属于内层作用域。下面的例子说明了这点：

```
a = -99


def outer():
    b = 35

    def inner():
```

```
            c = 2
            #该绑定屏蔽了外层本地变量b。
            b = 10
            #该绑定屏蔽了全局变量a。
            a = -20
            return abs(a + b*c)

        return inner()
```

请将上述代码保存为scope2.py，然后通过下面的命令行和语句验证：

```
$ python3 -i scope2.py

>>> outer()
0
>>>
```

上面的两个例子合在一起说明了这样一个事实：默认情况下，外层作用域中的变量对内层作用域来说是只读的，如果在内层作用域中对该变量执行绑定操作，则会导致在内层作用域中定义同名标识符，进而将外层作用域中的变量屏蔽。这种在外层作用域中定义，在内层作用域中读取的标识符，被称为“自由变量（free variables）”。对于上面例子中的inner()来说，a和b都是自由变量，而c则是它自己定义的本地变量。

对于没有明确指定其所属名字空间的标识符的判断发生在Python解释器进行第一轮处理时，而这轮处理不会实际执行任何语句，因此也不会进行真正的绑定。这就导致了无法发现标识符先被访问后被绑定的错误。请看下面的例子：

```
>>> a = 1
>>> def f():
...     print(a)
...     a = 2
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: cannot access local variable 'a' where it is not
associated with a value
>>>
```

对于该例子的解释是这样的：在执行定义f()的函数定义语句时，由于其函数体内包含关于a的赋值语句，因此第一轮处理完后a会被识别为本地变量。而函数定义语句执行完成后a并不会被绑定到2，只有“a是本地变量”这一结论被固化在代码对象中。接下来调用f()时，代码对象会被执行，而本地变量a的绑定是在其被输出之后的，因此抛出异常。整个过程都不涉及全局变量a。

有些时候，我们需要对自由变量进行写操作。这种情况下需要使用nonlocal语句和global语句。

nonlocal语句的语法为：

```
nonlocal identifier, ...
```

其功能是将列表中的标识符声明为外层本地变量。此后对于这些标识符将直接从倒数第二层作用域对应的名字空间开始搜索，到全局名字空间为止（但不会搜索全局名字空间）。如果在这个范围内没有找到指定的标识符，则抛出NameError异常。

global语句的语法为：

```
global identifier, ...
```

其功能是将列表中的标识符声明为全局变量。此后对于这些标识符将直接在全局名字空间中搜索，如果找不到则在内置名字空间中搜索。注意global语句中不能出现下列标识符：

- 通过函数定义语句定义的函数名。
- 通过类定义定义的类名。
- 函数的形式参数列表中的标识符。
- 被导入的模块名。
- try语句中的except子句、with语句、for语句和match语句中定义的标识符。

换句话说，global语句中仅能出现通过赋值语句或赋值表达式定义的标识符。

此外，Python解释器还提供了内置函数locals()和globals()供我们查看任意作用域视角的本地变量和全局变量。而第3章中介绍的内置函数dir()和vars()在不给出object参数时也可以达到类似的目的。

locals()的语法为：

```
locals()
```

它以字典的形式返回当前作用域中所有本地变量（包括自由变量）引用的对象。如果当前作用域对应着全局名字空间，则locals()等价于globals()。

globals()的语法为：

```
globals()
```

它以字典的形式返回当前全局名字空间中所有全局变量引用的对象，但不包括内置名字空间中的变量。

当vars()省略了object参数时，将与locals()完全等同。而当dir()省略了object参数时，将以列表的形式返回当前作用域中所有变量的名称，相当于只截取了locals()返回字典中的所有键。

下面的例子说明了local语句、global语句、locals()（以及等价的vars()）、globals()和dir()的用法：

```
a = 1
print('a = ' + str(a))


def f1():
    b = 2
    print('b = ' + str(b))

    def f2():
        c = 3
        print('c = ' + str(c))

        def f3():
            d = 4
            #将标识符b和c声明为本地变量中的自由变量。
            nonlocal b, c
            #将标识符a声明为全局变量。
            global a
            #直接写入f2()定义的本地变量c。
            c = -3
            #直接写入f1()定义的本地变量b。
            b = -2
            #直接写入全局变量a。
            a = -1
            print('\n')
            #列出f3()视角的所有本地变量名称。
            print('dir: ' + str(dir()))
            #显示f3()视角的所有本地变量状态。
            print('locals: ' + str(locals()))
            #显示f3()视角的所有全局变量状态。
            print('globals: ' + str(globals()))
            print('\n')

        f3()
        print('c = ' + str(c))

    f2()
    print('b = ' + str(b))

f1()
```

```
    print('a = ' + str(a))


    print('\n')
    #列出所有全局变量名称。
    print('dir: ' + str(dir()))
    #显示所有全局变量状态。
    print('locals: ' + str(locals()))
    #显示所有全局变量状态。
    print('globals: ' + str(globals()))
```

请将上述代码保存为scope3.py，然后通过下面的命令行验证：

```
    $ python3 scope3.py

    a = 1
    b = 2
    c = 3


    dir: ['b', 'c', 'd']
    locals: {'d': 4, 'b': -2, 'c': -3}
    globals: {'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <_frozen_importlib_external.SourceFileLoader object at
0x1048bc610>, '__spec__': None, '__annotations__': {}, '__builtins__': <module
'builtins' (built-in)>, '__file__': '/Users/wwy/scope3.py', '__cached__': None,
'a': -1, 'f1': <function f1 at 0x1047f7eb0>}


    c = -3
    b = -2
    a = -1


    dir: ['__annotations__', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__', '__spec__', 'a', 'f1']
    locals: {'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <_frozen_importlib_external.SourceFileLoader object at
0x1048bc610>, '__spec__': None, '__annotations__': {}, '__builtins__': <module
'builtins' (built-in)>, '__file__': '/Users/wwy/scope3.py', '__cached__': None,
'a': -1, 'f1': <function f1 at 0x1047f7eb0>}
    globals: {'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <_frozen_importlib_external.SourceFileLoader object at
0x1048bc610>, '__spec__': None, '__annotations__': {}, '__builtins__': <module
'builtins' (built-in)>, '__file__': '/Users/wwy/scope3.py', '__cached__': None,
'a': -1, 'f1': <function f1 at 0x1047f7eb0>}
```

6-3. 执行动态生成的代码

（标准库：内置函数）

就像很多其他解释型语言一样，Python也提供了将一个字符串当成代码动态地解释执行的接口，即内置函数compile()、eval()和exec()。下面依次讨论这三个内置函数。

compile()的功能是基于指定的代码通过伪编译生成一个代码对象，其语法为：

```
compile(source, filename, mode, flags=0, dont_inherit=False, optimize=-1)
```

其中source参数用于传入被当成代码的字符串。事实上source也可以被传入一个AST对象，但本书不讨论，有兴趣的读者可查阅标准库中的ast模块。

filename参数一般被传入一个指向包含被传入字符串的源文件的路径，但当代码是动态生成时可被传入类似于“<string>”的字符串。不论哪种情况被传入filename的字符串都会被compile()返回的代码对象的co_filename属性引用。

mode参数决定了伪编译代码时采用哪种模式，具体可传入如下对象：

- “single”：将代码当成以交互式执行的单独一条语句来编译。这样当生成的代码对象被执行时，如果有返回值，则直接将返回值写入标准输出。
- “eval”：将代码当成一个表达式来编译。生成的代码对象应通过eval()执行。
- “exec”：将代码当成一条或多条语句来编译。生成的代码对象应通过exec()执行。

flags参数和dont_inherit参数共同控制着应当激活哪些编译选项，以及应当允许哪些future特性。有三种情况：

- 如果flags被传入0且dont_inherit被传入False，则仅使用编译调用compile()的代码本身时使用的编译选项设置。
- 如果flags被传入正整数且dont_inherit被传入False，则以编译调用compile()的代码本身时使用的编译选项设置为基础，根据flags进行修改。
- 如果dont_inherit被传入True，则仅使用flags指定的设置，哪怕flags被传入0。

本书不详细讨论编译选项和future特性，有兴趣的读者可查阅标准库中的ast模块和__future__模块。

optimize参数用于指定compile()的优化级别，可传入如下对象：

- -1：与Python解释器启动时的优化级别设置保持一致。
- 0：不做任何优化，保留assert语句，将__debug__设置为True。
- 1：相当于添加了-O，移除assert语句，将__debug__设置为False。
- 2：相当于添加了-OO，在-O的基础上，额外丢弃文档字符串。

由于compile()几乎总是与eval()或exec()联合使用，所以这里不给出例子。

---

eval()和exec()在运行机制上类似，但存在微妙的差别。下面先介绍eval()，其语法为：

```
eval(expression[, globals[, locals]])
```

其三个参数的含义是：

➤ expression：传入一个被当成表达式求值的字符串，也可以传入一个代码对象。
➤ globals：传入一个代表全局名字空间的字典。如果被省略，则自动传入对应eval()被调用时的全局名字空间的字典，即globals()返回的对象。
➤ locals：传入一个储存有全部本地变量（包括自由变量）的映射。如果被省略，则自动传入包含由调用eval()的语句所在作用域确定的全部本地变量的字典，即locals()返回的对象。

eval()会返回将字符串当成表达式求值得到的对象。

绝大多数情况下，使用eval()时不会向globals参数和locals参数传入对象。一个可能的场景是给globals参数传入一个字典，该字典的__builtins__键引用的对象是对应内置名字空间的模块的一个子模块，这样就限制了对传入expression参数的字符串求值时能够使用的内置函数、内置常量、内置类型和内置异常。特别的，如果给globals参数传入了一个字典，该字典不具有__builtins__键，则Python解释器会自动给该字典添加该键，并使其引用对应内置名字空间的模块。

下面的例子说明了eval()的作用：

```
import re

a = 15
b = 6
c = 79

while True:
    expr = input("请输入一个以a、b和c中的两个作为参数的四则运算式（输入空串终止）：")
    if expr:
        if re.match(r"^[abc][\+\-\*\/][abc]$", expr):
            d = eval(expr)
            print(expr + "的结果为" + str(d) + "。")
        else:
            print("请输入正确的运算式！")
    else:
        break
```

请将上述代码保存为eval.py，然后执行该脚本：

```
$ python3 eval.py
请输入一个以a、b和c中的两个作为参数的四则运算式（输入空串终止）：a+b
a+b的结果为21。
请输入一个以a、b和c中的两个作为参数的四则运算式（输入空串终止）：b/c
b/c的结果为0.0759493670886076。
请输入一个以a、b和c中的两个作为参数的四则运算式（输入空串终止）：c*a
c*a的结果为1185。
请输入一个以a、b和c中的两个作为参数的四则运算式（输入空串终止）：c-d
请输入正确的运算式！
请输入一个以a、b和c中的两个作为参数的四则运算式（输入空串终止）：a**b
请输入正确的运算式！
请输入一个以a、b和c中的两个作为参数的四则运算式（输入空串终止）：
```

上面的例子说明了使用eval()时需要注意的两点。第一点，如果不给globals参数和locals参数传入对象，那么通过expression参数传入的字符串就相当于嵌入了eval()所在的位置被执行，因此可以访问在该字符串之外定义的标识符。第二点，eval()具有执行用户输入的字符串的能力，因此存在潜在的安全隐患，使用时必须特别小心，例如像上面的例子中那样通过正则表达式验证输入的字符串符合指定的模式才执行。

而下面的例子说明了如何联合使用compile()与eval()：

```
$ python3

>>> import random
>>> code = compile('random.randint(a, b)', '<string>', 'eval')
>>> a = 1
>>> b = 5
>>> eval(code)
4
>>> eval(code)
3
>>> eval(code)
1
>>> a = 6
>>> b = 10
>>> eval(code)
10
>>> eval(code)
9
>>> eval(code)
9
>>>
```

---

exec()的语法与eval()非常相似：

```
exec(object[, globals[, locals]])
```

其中的globals参数和locals参数的含义与在eval()中完全相同。exec()与eval()的区别是如下两点：

➤ object参数被传入的字符串或代码对象可以比一个表达式复杂得多，甚至可以包含函数定义和类定义。然而如果这段代码包含nonlocal语句，return语句或yield语句，那么它们只能出现在函数定义的函数体中。
➤ exec()的返回值永远是None。

下面对之前的例子进行改造，以exec()替换掉eval()以实现相同的功能：

```
import re

a = 15
b = 6
c = 79

while True:
    expr = input("请输入一个以a、b和c中的两个作为参数的四则运算式（输入空串终止）：")
    if expr:
        if re.match(r"^[abc][\+\-\*\/][abc]$", expr):
            code = 'print("' + expr + '的结果为" + str(' + expr + ') +
"。")'
            exec(code)
        else:
            print("请输入正确的运算式！")
    else:
        break
```

请将上述代码保存为exec.py，然后验证它与eval.py的功能相同。

下面的例子则说明了如何联合使用compile()与exec()：

```
$ python3

>>> import random
>>> code = compile('''
... def f(a, b):
...     return random.randint(a, b)
... ''', '<string>', 'exec')
>>> exec(code)
>>> a = 1
>>> b = 5
>>> f(a, b)
2
>>> f(a, b)
5
>>> f(a, b)
3
>>> a = 6
>>> b = 10
>>> f(a, b)
10
>>> f(a, b)
10
>>> f(a, b)
```

```
6
>>>
```

最后需要强调，不论是eval()还是exec()，在执行过程中都会给函数栈额外添加一个特殊帧对象。而eval()和exec()被传入的字符串或代码对象并不属于调用它们的模块，而属于一个动态生成的临时模块。

6-4. 函数的闭包

（语言参考手册：3.2）
（标准库：types）

在第4章中讨论函数装饰器时曾经提到过“闭包（closures）”这一概念。这里明确给出闭包的定义：如果一个函数在另一个函数的函数体内定义，则前者就是后者的闭包。闭包的核心特性是能够访问包含它的函数的本地变量，且可以在后者某次调用返回后依然能访问到这些本地变量当时引用的对象，就好像保存了这些本地变量当时状态的一个快照。

下面给出一个闭包的典型例子：

```
#定义外层函数f1。
def f1(a, b):
    x = a
    y = b
    #定义内层函数f2。
    def f2(z):
        return (x, y, z)
    #该return语句使f2能被其他标识符引用。
    return f2
```

请将上述代码保存为closure1.py，然后通过如下命令行和语句验证：

```
$ python3 -i closure1.py

>>> l1 = f1(10, 20)
>>> l1(3)
(10, 20, 3)
>>> l1(5)
(10, 20, 5)
>>> l2 = f1(-9, 0)
>>> l2(100)
(-9, 0, 100)
>>> l1(-3)
(10, 20, -3)
>>>
```

上面的例子有较形象的几何解释：外层函数f1()的两个参数a和b用于定位到三维空间中x-y平面上的一个点(x, y)，返回的内层函数f2()则用于在过该点垂直于x-y平面的直线上扫描，其

参数z代表直线上某点的z坐标。而对f1()的每次调用返回的f2()都会“记录下”该次调用中参数a和参数b被传入的数字，该记录行为在代码中并没有体现。这就是闭包最神奇的地方。

事实上，闭包只是利用作用域规则实现的一种技巧。

为了理解闭包，必须进一步了解作用域规则的底层实现机制。

下面考虑形成闭包的条件——内层函数的定义在外层函数的函数体内。这会对内层函数造成什么影响呢？

第4章已经说明，函数定义语句被执行时只会创建代码对象，由相应函数对象的__code__特殊属性引用。当函数被调用时才会创建帧对象，其f_locals属性最初引用一个空字典，然后按照第4章描述的方法在其内创建“arguments”列表来存放实际参数，最后在执行代码对象的过程中，每遇到一个本地变量绑定操作，就向该字典中添加相应的键值对。这解释了函数如何处理自己定义的本地变量。然而这一机制并不能用于处理自由变量。

上一节已经说明，自由变量包括全局变量和外层作用域中的本地变量，而全局变量又分为在全局名字空间中和在内置名字空间中两种情况。内置名字空间总是可以访问，且其内的标识符引用的对象是固定不变的。由于函数定义所在脚本和函数调用所在脚本可以不是一个（后者通过import语句从前者导入了该函数），函数对象必须记录下它被创建时的全局名字空间，而这是通过其特殊属性__globals__实现的。当帧对象被创建时，其f_globals属性直接拷贝函数对象的__globals__，以使得全局名字空间中的全局变量可被访问。

而对于外层作用域中的本地变量，函数对象则是通过特殊属性__closure__记录的，该属性引用的是一个“单元对象（cell objects）”形成的元组。标准库中的types模块提供了types.CellType来代表单元对象类型。每个单元对象都通过其cell_contents属性引用一个对象，它其实是该函数对象被创建时外层作用域中的某个本地变量引用的对象。当对外层函数的调用返回时，其部分本地变量引用的对象依然被内层函数的单元对象引用，因此不会被垃圾回收机制销毁，进而形成了这些本地变量当时状态的快照。

当内层函数被调用时，新创建的帧对象的f_locals属性引用的字典会被自动填入所有单元对象引用的对象，以使它们能被访问。这也是通过locals()或vars()查看一个作用域中的本地变量时，结果中会包括非全局自由变量的原因。当然，如果一个函数的函数体内没有任何非全局自由变量，其__closure__特殊属性将引用None。

继续上面的例子，请通过如下语句验证闭包的__closure__记录了哪些对象：

```
     >>> type(f1.__closure__)
     <class 'NoneType'>
     >>> l1.__closure__
     (<cell at 0x102cdb040: int object at 0x102b68210>, <cell at 0x102cdb010:
int object at 0x102b68350>)
```

```
>>> l1.__closure__[0].cell_contents
10
>>> l1.__closure__[1].cell_contents
20
>>> l2.__closure__
(<cell at 0x102cfc1c0: int object at 0x102e500d0>, <cell at 0x102cfc340:
int object at 0x102b680d0>)
>>> l2.__closure__[0].cell_contents
-9
>>> l2.__closure__[1].cell_contents
0
>>>
```

在明白了闭包的原理后，就不难理解那些关于它们的反直觉例子了。下面给出一个较典型的例子：

```
def outer():
    #定义一个用于储存闭包的列表。
    inner = list()
    n = 3
    while n > 0:
        #循环三次，每次都产生一个返回自由变量n的闭包。
        def f():
            return n
        inner.append(f)
        n = n-1
    return inner
```

请将上述代码保存为closure2.py，然后通过如下命令行和语句验证：

```
$ python3 -i closure2.py

>>> i = outer()
>>> i[0]()
0
>>> i[1]()
0
>>> i[2]()
0
>>>
```

调用outer()返回的列表包含3个闭包，但调用它们得到的结果都是0，而非直觉中的3、2和1。这是因为这3个闭包的__closure__引用的三个元组都包含同一个单元对象，该单元对象的cell_contents属性指向的是循环变量n，而在outer()返回时n已经变成了0。请通过如下语句验证：

```
>>> i[0].__closure__
(<cell at 0x1050370d0: int object at 0x104ec80d0>,)
>>> i[1].__closure__
(<cell at 0x1050370d0: int object at 0x104ec80d0>,)
>>> i[2].__closure__
```

```
    (<cell at 0x1050370d0: int object at 0x104ec80d0>,)
    >>>
```

至此我们讨论完了闭包。理论上，类体之间也可以嵌套，函数体和类体之间也可以互相嵌套。那么当有类参与时，是否存在类似闭包的概念呢？

请注意，上节介绍作用域规则时，仅考虑了类体被全局名字空间包含的情况，有意避开了类体之间互相嵌套，以及函数体和类体之间互相嵌套的情况，这是因为在这些情况下上节介绍的作用域规则将不再适用。

首先考虑类体相互嵌套的情况。下面的语句说明类体之间是可以嵌套的，不算语法错误：

```
>>> class A:
...     class B:
...         pass
...
>>> obj1 = A()
>>> type(obj1)
<class '__main__.A'>
>>>
>>> obj2 = A.B()
>>> type(obj2)
<class '__main__.A.B'>
>>>
```

然而下面的语句则说明，类体中的自由变量只能是全局变量，不能是外层类体的本地变量：

```
>>> n = 0
>>> class A:
...     n = 1
...     class B:
...         m = n
...
>>> A.n
1
>>> A.B.m
0
>>>
```

接下来考虑类体内包含函数体的情况。请注意，这些函数其实就是函数类属性，但在第5章中没有涉及它们的作用域规则。下面的例子说明不论是方法、类方法还是静态方法，其函数体内的自由变量也都只能是全局变量，不能是外层类体的本地变量：

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> class A:
...     a = 10
...     b = 20
...     c = 30
...     def geta(self):
...         return a
...     @classmethod
...     def getb(cls):
...         return b
...     @staticmethod
...     def getc():
...         return c
...
>>> obj = A()
>>> obj.geta()
1
>>> obj.getb()
2
>>> obj.getc()
3
>>>
```

最后考虑函数体内包含类体的情况。下面的语句说明，当类体被函数体包含时，类体内的自由变量可以是函数体的本地变量：

```
>>> def f():
...     x = 'Hello'
...     class H:
...         h = x
...     return H
...
>>> H = f()
>>> H.h
'Hello'
>>>
```

类对象与函数对象具有如下三点不同：

- 类对象没有__closure__特殊属性，无法记录外层作用域中的本地变量。
- 类对象没有__globals__特殊属性，无法记录类对象被创建时的全局名字空间。
- 类体只会被执行一次，结果是创建该类对象。此后类对象被调用时与类体无关，因此类对象也没有引用对应类体的代码对象的__code__特殊属性。

这三点不同使类体的作用域是相对独立的，具有特殊的作用域规则：从类体内访问外层作用域中的标识符时，和函数体的规则相同；而当类体本身作为外层作用域时，就仿佛它并不存在。下面用该作用域规则解释上面三个例子。

对于第一个例子，在类A中对n的写操作会创建本地变量n，这符合函数作用域规则。而在类B中对n的读操作则跳过了类A，直接从全局名字空间中读取，这是类作用域独有的。

对于第二个例子，在geta()、getb()和getc()中分别访问a、b和c，同样跳过了类A，直接从全局名字空间中读取，这也是类作用域独有的。该例子同样说明了如下三点：

- 在方法中访问实例属性和类属性都必须通过self。
- 在类方法中访问类属性必须通过cls。
- 在静态方法既无法访问实例属性，也无法访问类属性。

对于第三个例子，在类H中对x的读操作是从外层函数f()的函数体中开始搜索的，这同样符合函数作用域规则。

6-5. 文档字符串

（教程：4.7、4.8.7、4.9）
（标准库：内置函数）

至此相信你已经掌握了如何将三种构建单元组合在一起构建出完整的程序。这个程序可以任意复杂，尽管你可以通过注释来解释代码的含义，但用户不会愿意阅读源代码和其内的注释，甚至与你参与同一个项目的合作者都会觉得阅读注释的效率不高。为了解决这一问题，Python提供了“文档字符串（docstrings）”这种机制，并结合内置函数help()（以及其他第三方文档字符串处理工具）为你的程序自动生成易于阅读的文档。

文档字符串指的是符合下列标准之一的字符串：

- 该字符串被作为函数体内的第一条语句。
- 该字符串被作为类体内的第一条语句。
- 该字符串被作为Python脚本的第一条语句。

它们会分别成为函数、类和模块的文档字符串，被相应函数对象、类对象和模块对象的特殊属性__doc__引用。

文档字符串的语法规则很简单，但要用好它们却并不容易。简单来说，文档字符串是对函数、类和模块功能的总结，既不是它们的接口声明（这可以通过函数定义语句和类定义语句本身来反映），也不是它们的实现细节（这些细节应通过注释来描述）。PEP 257[1]给出了文档字符串内容和格式的惯例，所有Python程序员都应该遵循这些惯例。

- 文档字符串应该用三引号"""括起来，不论它只有一行还是具有多行。在绝大多数情况下，文档字符串中只应包括ASCII字符，以确保能在任何设备上正常显示。如果你确实需要在

文档字符串中包含ASCII之外的Unicode字符，则应在第一个三引号之前添加“u”，即写为u"""，以表明这是一个Unicode文档字符串。

➤ 一般而言，文档字符串应该包含多行。第一行是对构建单元的一句话描述，需紧跟第一个"""，其间不要有空格。第二行必须是一个空行。剩下的行是更详细的描述，亦即对第一行的补充。这些剩下的行的行首可以有任意多个空格，但其中第一行的行首空格数必须最小。当help()或其他文档字符串处理工具对文档字符串进行处理时，会对第一行之外的非空行进行如下处理：若这些行中的第一行的行首空格数为n，则所有这些行的行首空格数都减去n。

➤ 如果一个多行文档字符串描述的是类，则在其后应跟随一个空行，以将它与定义类属性的语句分隔开。描述模块的多行文档字符串和后续语句之间应该有一个（后续语句是import语句）或两个（后续语句是函数定义语句或类定义语句）空行。而描述函数的多行文档字符串不需要用空行与后续语句隔开。

➤ 只有一行的文档字符串应以第二个"""作为行尾，且在任何情况下都不需要用空行将其与后续语句隔开。

文档字符串应包含如下内容：

1. 函数文档字符串需总结该函数的行为（可以包括参数和返回值的类型），副作用和可能抛出的异常，以及调用时的限制。
2. 类文档字符串应列出该类包含的公有类属性（特别注明对基类进行重写的类属性），以及该类实例的公有实例属性。类实例化时需要的初始化参数列表则应通过该类的__init__的函数文档字符串描述。
3. 模块文档字符串应列出该模块定义的类和函数。

下面的例子显示了各种情况下的文档字符串：

```
    """This module illustrates how docstrings work.

        In this module, one function f and one class C are defined, each of
which has a docstring. C also has one method, which also has a docstring. And
here is the docstring for the whole module.
    """


    def f():
        """This function does nothing. """
        pass


    class C():
        """This class does nothing.
```

```
        It has no public methods and attributes.

        And no public instance attributes are defined.
    """

    def __init__(self):
        """When initializing this class, no arguments are needed."""
        pass
```

请将上述代码保存为docstring.py。

接下来我们讨论如何通过内置函数help()来使用文档字符串（第三方文档字符串处理工具超出了本书的范围）。help()的语法为：

```
help([object])
```

它总是在Python解释器的交互模式下被调用，功能是启动Python解释器内置的帮助系统。

若调用help()时省略了object参数，则会启动交互式帮助系统，提示符会变成“help>”。此时可以通过输入模块名、类名、函数名、变量名、关键字……等来查看对应文档。当然在默认情况下只能查看关于标准库中的模块，以及内置函数、内置常量、内置类型和内置异常的文档。文档会以Unix手册页的形式被显示，按下q退出查看。特别的，输入“quit”将退出交互式帮助系统。请通过如下语句验证help()的这种语法（单独的“q”代表按下q退出文档查看）：

```
>>> help()
help> types
q
help> type
q
help> id
q
help> TypeError
q
help> def
q
help> quit
>>>
```

调用help()时给object参数传入了一个名字，则会直接显示相关文档，即“help(name)”等价于在“help>”提示符下输入“name”。但标准库中的模块需要先导入然后才能查看其文档。请通过如下语句验证help()的这种语法：

```
>>> import types
>>> help(types)
q
>>> help(type)
q
>>> help(id)
```

```
q
>>> help(TypeError)
q
>>>
```

易知，如果想要用help()查看我们自己编写的Python脚本中的文档，就需要先导入相应模块。请通过如下语句验证在交互式帮助系统中查看docstring.py的文档：

```
>>> import docstring
>>> help()
help> docstring
q
help> docstring.C
q
help> docstring.C.__init__
q
help> doctoring.f
q
help> quit
>>> ^D
```

我们也可以通过执行脚本来定义相关类和函数，然后再查看关于它们的文档。但要注意，这样做会导致这些函数和类被视为在主模块中定义的，且无法查看关于脚本本身的文档。请通过如下命令行和语句验证以这种方式查看docstring.py的文档：

```
$ python3 -i docstring.py
>>> help(C)
q
>>> help(C.__init__)
q
>>> help(f)
q
>>>
```

接下来让我们看看help()是如何基于文档字符串生成文档的。如下是关于模块docstring的文档：

```
Help on module docstring:

NAME
    docstring - This module illustrates how docstrings work.

DESCRIPTION
    In this module, one function f and one class C are defined, each of
which has a docstring. C also has one method, which also has a docstring. And
here is the docstring for the whole module.

CLASSES
    builtins.object
        C
```

```
    class C(builtins.object)
     |  This class does nothing.
     |
     |  It has no public methods and attributes.
     |
     |  And no public instance attributes are defined.
     |
     |  Methods defined here:
     |
     |  __init__(self)
     |      When initializing this class, no arguments are needed.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    f()
        This function does nothing.
FILE
    /Users/wwy/docstring.py
```

如下是关于类C的文档：

```
Help on class C in module docstring:

docstring.C = class C(builtins.object)
 |  This class does nothing.
 |
 |  It has no public methods and attributes.
 |
 |  And no public instance attributes are defined.
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      When initializing this class, no arguments are needed.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

如下是关于类C中的__init__的文档：

```
Help on method __init__ in module docstring:

docstring.C.__init__ = __init__(self)
    When initializing this class, no arguments are needed.
```

如下是关于函数f的文档：

```
Help on function f in module docstring:

docstring.f = f()
    This function does nothing.
```

可以看出，除了文档字符串外，help()在生成文档时还收集了其他有用的信息，包括模块名、类名、函数名和参数等等。事实上help()是通过标准库中的pydoc模块实现其功能的，但对pydoc模块的讨论超出了本书的范围。

PEP 8推荐给所有脚本、类和函数都添加文档字符串，以方便别人和作者自己以后查看。

6-6. 包

（教程：6.4）
（语言参考手册：5.2）

如果你满足于每个模块对应一个文件，相互间都是平行的，使用时全部放在工作目录下，那么上面介绍的技巧已经足够了。然而当你参与较大的项目时，就会发现这种组织方式难以接受：所有Python脚本都放在一个目录下会极大地提高管理难度和出错的概率。我们希望通过目录将大量Python脚本按照逻辑分门别类地整理起来，然而前面介绍的import语句的使用方法使我们必须将所有相关目录添加到PYTHONPATH环境变量或sys.path，相当不方便。

为了解决这一问题，Python引入了“包（packages）”。包其实就是一种特殊的模块，对应它们的模块对象具有__path__特殊属性，以支持在其内搜索其他模块。根据__path__特殊属性引用对象的区别，包又分为“常规包（regular packages）”和“名字空间包（namespace packages）”两类。名字空间包是Python3.3引入的，是对常规包的扩展。本书将详细讨论常规包，仅简略介绍名字空间包的概念。

常规包技术使Python解释器可以基于文件系统中的目录创建模块对象。然而并非所有目录都能被视为常规包，前提条件是该目录下有名为“__init__.py”的脚本，该脚本会在创建相应模块对象时被自动执行。

请在工作目录下创建名为“dir1”的目录，并在该目录下创建一个空的__init__.py文件。这样就创建了dir1包，其结构为：

```
└─ dir1/
    └─ __init__.py
```

请通过如下命令行和语句验证dir1包的特殊属性：

```
$ python3

>>> import dir1
>>> dir1
<module 'dir1' from '/Users/wwy/dir1/__init__.py'>
>>> dir1.__name__
'dir1'
>>> dir1.__file__
'/Users/wwy/dir1/__init__.py'
>>> dir1.__path__
['/Users/wwy/dir1']
>>> dir1.__cached__
'/Users/wwy/dir1/__pycache__/__init__.cpython-311.pyc'
>>> dir1.__package__
'dir1'
>>>
```

注意上述结果中的“/Users/wwy/”代表我的工作目录（即macOS X下的家目录），它会根据你选择的工作目录而变化。

该结果说明，常规包对应的模块对象依然是基于文件创建的，即__init__.py文件，只不过模块名变成了目录名。事实上，常规包的文档字符串也是在__init__.py中给出的，格式与模块的文档字符串完全相同，但应包括如下内容：该包的所有模块、子模块和子包。

但要注意，常规包的__cached__特殊属性不再引用None，而是引用指向__init__.py文件经伪编译得到的.pyc文件的路径；而常规包的__package__实例属性也不再引用None（因为此时已经有包这一概念存在），在上面的例子中dir1的__pacakge__引用了包名“dir1”。

此外，常规包的__path__特殊属性引用一个字符串列表，每个字符串都是指向文件系统中某个目录的路径。当在该常规包中搜索模块时，Python解释器会在该列表指定的目录下搜索。由于代表常规包的模块对象也是动态生成的，所以储存在__path__中的都是绝对路径，反映了模块对象被创建时常规包在文件系统中的位置。__path__默认只会储存一个绝对路径，指向常规包对应的目录本身。

常规包自身也可以定义标识符，只需要将相应的赋值语句、函数定义语句和类定义语句放在__init__.py中即可。然而一般不会这样做，而是将标识符的定义放在其他Python脚本中，然后把这些脚本放在对应常规包的目录下（即和__init__.py在同一个目录）。这些脚本是该常规包所包含的模块，可以通过“package.module”的形式被定位，且相应模块对象的实例属性__package__会引用该包的包名。这样做的优点是，若我们对当前打包方式不满意，只需在文件系统中移动相关Python脚本就能改变包的结构，而__init__.py只需要做简单的调整即可。

有了包之后，import语句的module参数和relative_module参数既能以“package”的形式导入整个包，也能以“package.module”的形式导入该包中的某个指定模块。特别的，relative_module参数还可以是“.module”形式的相对模块名，并被解读为当前包中的某个指定模块，这就是它与module参数的区别。

首先考虑将import语句的第一种语法应用于包。由于导入包中的某个模块时需要先导入该包，所以该包的__init__.py总是会被执行，而包名也总是会被作为标识符导入。可以通过__init__.py中的import语句来控制哪些标识符被视为包的基本组成部分，这些标识符在这种情况下能够以“package.attribute”的形式访问。

对于包中的模块来说，不论是其模块名还是其内定义的某个标识符成为了包的基本组成部分（亦即在__init__.py中通过import语句导入），该模块就变得与包不可分割了：只要导入包，就会自动为该模块创建模块对象，同时也自动导入该模块的模块名。这意味着只要导入了包，就能以“package.module.attribute”的形式访问该模块定义的所有标识符，不论它是否是包的基本组成部分。对于一个包内模块来说，如果它的模块名和它定义的任何标识符都不是包的基本组成部分，那它就是包的一个“子模块（sub modules）”。子模块仅在通过额外的import语句被导入时才会创建对应的模块对象。

下面用一个例子来验证上述论断。请在dir1目录下创建a.py和b.py，内容分别为：

```
A1 = 1
A2 = 2
```

和

```
B3 = 3
B4 = 4
```

然后将dir1目录下的__init__.py的内容设置为：

```
from .a import A1
```

此时dir1包的结构是：

```
└─ dir1/
    ├─ __init__.py
    ├─ a.py
    └─ b.py
```

让我们看看导入dir1包后能得到哪些标识符。请执行如下命令行和语句：

```
$ python3

>>> import dir1
>>> dir1.A1
1
>>> dir1.A2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'dir1' has no attribute 'A2'. Did you mean: 'A1'?
>>> dir1.B3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'dir1' has no attribute 'B3'
>>> dir1.B4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'dir1' has no attribute 'B4'
>>> dir1.a
<module 'dir1.a' from '/Users/wwy/dir1/a.py'>
>>> dir1.a.__package__
'dir1'
>>> dir1.a.A2
2
>>> dir1.b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'dir1' has no attribute 'b'
>>>
```

该结果说明，导入包dir1后能够直接通过“dir1”访问到的标识符只有A1，然而定义A1的模块a也自动被导入，因此可以通过“dir1.a”访问到的标识符有A1和A2。模块b和其内定义的标识符B3和B4则都不能访问，意味着模块b是包dir1的子模块。

接下来让我们看看导入dir1.b模块后能得到哪些标识符。请执行如下命令行和语句：

```
$ python3

>>> import dir1.b
>>> dir1.b
<module 'dir1.b' from '/Users/wwy/dir1/b.py'>
>>> dir1.b.__package__
'dir1'
>>> dir1.b.B3
3
>>> dir1.b.B4
4
>>> dir1
<module 'dir1' from '/Users/wwy/dir1/__init__.py'>
>>> dir1.a
<module 'dir1.a' from '/Users/wwy/dir1/a.py'>
>>> dir1.A1
1
>>> dir1.a.A2
2
```

```
>>>
```

该结果说明，Python解释器在为模块b创建模块对象之前，会先为包dir1创建模块对象，然后执行__init__.py，导致为模块a创建模块对象。这证明了模块a与包dir1不可分割。

如果只想导入包内的某个模块，而不想导入包本身，那么应在import语句中使用as子句为导入的模块重命名。请执行如下命令行和语句：

```
$ python3

>>> import dir1.b as mb
>>> mb.__package__
'dir1'
>>> mb.B3
3
>>> mb.B4
4
>>> dir1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'dir1' is not defined. Did you mean: 'dir'?
>>>
```

事实上，以这种方式导入模块dir1.b时依然为包dir1和模块dir1.a创建了模块对象，并执行了__init__.py，然而对应dir1和dir1.a的模块对象没有被绑定到任何标识符，因此无法被访问。（注意模块对象的__package__实例属性引用的是代表包名的字符串，并非对应包的模块对象本身。）然而这些模块对象并不会被销毁，因为它们其实还被sys.module引用（会在本章最后详细说明）。

接下来考虑将import语句的第二种语法应用于包。请执行如下命令行和语句：

```
$ python3

>>> from dir1 import A1
>>> A1
1
>>> from dir1.a import A2
>>> A2
2
>>> from dir1.b import B3, B4
>>> B3
3
>>> B4
4
>>>
```

该结果说明从模块导入标识符时，包不会造成额外影响，唯一需要注意的是包自身也可以被视为一个标识符来源。

那么，将import语句第三种语法应用于包会有什么效果呢？请执行如下命令行和语句：

```
$ python3

>>> from dir1 import *
>>> a
<module 'dir1.a' from '/Users/wwy/dir1/a.py'>
>>> A1
1
>>> A2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'A2' is not defined. Did you mean: 'A1'?
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>> B3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'B3' is not defined
>>> B4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'B4' is not defined
>>>
```

该结果说明，默认情况下从一个常规包导入标识符时，等价于导入其__init__.py中导入的标识符，而并非导入该包所包含的所有模块定义的所有标识符。特别的，在__init__.py中添加类似如下的语句，可以使得将import语句的第三种语法应用于包时，自动导入该包包含的所有模块：

```
__all__ = ['a', 'b']
```

至此已经讨论完了常规包的基本逻辑。容易看出，包与其内模块的关系，与目录与其内文件的关系是类似的。进一步，就像目录可以包含子目录一样，包也可以包含“子包（sub packages）”，而这就是接下来讨论的话题。首先强调一点，对应子包的模块对象的实例属性__package__将引用父包的包名，与父包所包含的模块一样。

现在让我们在dir1目录下创建dir2目录，再在该目录下创建脚本c.py，内容为：

```
C5 = 5
C6 = 6
```

而该目录下的__init__.py的内容设置为：

```
from .c import C5, C6
```

这样dir2就成为了dir1的子包。接下来在dir2目录下创建dir3目录，再在该目录下创建脚本d.py，内容为：

```
D7 = 7
D8 = 8
```

而该目录下的__init__.py的内容设置为：

```
from .d import D7, D8
```

这样dir3就成为了dir2的子包。于是dir1包的结构变成了：

```
└─ dir1/
    ├─ __init__.py
    ├─ dir2/
    │    ├─ __init__.py
    │    ├─ dir3/
    │    │    ├─ __init__.py
    │    │    └─ d.py
    │    └─ c.py
    ├─ a.py
    └─ b.py
```

下面通过如下命令行和语句来验证将import语句的前两种语法应用到子包上的结果：

```
$ python3

>>> import dir1.dir2.dir3 as m3
>>> m3.D7
7
>>> m3.D8
8
>>> from dir1.dir2 import C5
>>> C5
5
>>>
```

最后，让我们给dir1包再添加一个子包dir4。dir4目录包含脚本e.py，其内容为：

```
E9 = 9
E10 = 10
```

此外，该目录下还有一个子目录dir5，包含脚本f.py，其内容为：

```
F11 = 11
F12 = 12
```

然而dir5目录下没有__init__.py，因此不是dir4的子包；而dir4目录下的__init__.py的内容设置为：

```
position = __path__[0]
__path__ = [position, position + '/dir5']

from .e import E9, E10
from .f import F11, F12
from ..dir2.dir3 import D7
```

现在dir1包的结构变为：

```
└─ dir1/
    ├─ __init__.py
    ├─ dir2/
    │   ├─ __init__.py
    │   ├─ dir3/
    │   │   ├─ __init__.py
    │   │   └─ d.py
    │   └─ c.py
    ├─ dir4/
    │   ├─ __init__.py
    │   ├─ dir5/
    │   │   └─ f.py
    │   └─ e.py
    ├─ a.py
    └─ b.py
```

下面通过如下命令行和语句来验证将import语句的第三种语法应用到子包上的结果：

```
$ python3

>>> from dir1.dir4 import *
>>> E9
9
>>> E10
10
>>> F11
11
>>> F12
12
>>> D7
7
>>>
```

在最后这个例子中，__init__.py中的代码使用了更多技巧。下面让我们集中探讨一下编写__init__.py需要注意的事项。

➤ 第一，可以在__init__.py中手工修改了__path__引用的列表。上面的例子使得在包dir4下搜索模块时，除了在目录dir4下搜索外，还在该目录的子目录dir5下搜索。这一技巧使我们能够将一个包所包含的模块继续基于目录分类，却不引入额外的子包。

➤ 第二，出现在__init__.py中的import语句几乎总是使用第二种语法，以明确导入哪些标识符到当前包。如果relative_module参数是“.”开头的相对包名/模块名，那么子包在包中移动后该语句依然有效。举例来说，即便dir4目录的位置发生了变化，语句“from .e import E9, E10”和“from .f import F11, F12”都不需要修改。

➤ 第三，通过让relative_module参数为“..”、“...”、……等多个“.”开头的相对包名/模块名，可以定位到该子包的父包、祖先包和姊妹包。在上面例子中，“..dir2.dir3”先定位到了dir4的姊妹包dir2，然后定位到了dir2的子包dir3。但要注意，如果子包在包中移动了，那么具有这样的relative_module参数的import语句都需要修改。

➤ 第四，如果__init__.py没有定义或导入任何标识符（例如它是一个空文件），则意味着该常规包只是用来访问子模块/子包的，自身没有包含任何标识符。

至此我们讨论完了常规包。下面简略介绍一下名字空间包。

名字空间包是对常规包的扩展，其目的是使一个包所包含的模块、子模块和子包不需要被限制在一个文件系统中，而可以遍布多个文件系统，甚至整个Internet。

代表名字空间包的模块对象的__path__特殊属性引用的是一个_NamespacePath类型的可迭代对象，每次迭代都产生一个搜索位置的条目，该条目可以是某文件系统中的路径、URL、数据库查询语句，甚至可以是将来发明的某种定位机制。

名字空间包不一定具有__init__.py。

本书不再进一步讨论名字空间包，感兴趣的读者请参阅PEP 420[2]。

最后，在明白了什么是包之后，我们可以给出“限定名（qualified names）”这个概念了。

一个限定名就是一个点号分隔的名字序列，提供了定位到某对象的信息。模块的限定名给出了它所属的包、该包所属的父包……依此类推。如果该限定名以“.”开头，则被视为“相对限定名（relative qualified names）”；否则，该限定名被视为“完全限定名（fully qualified names）”。

除了模块之外，函数和类也都有限定名。具体来说，在模块中直接定义的函数/类的函数名/类名就是它们的限定名后缀，而在类中定义的函数（即方法）的限定名后缀具有“clsname.funcname”的形式。而它们的限定名则是将各自的限定名后缀添加到所在模块的限定名之后得到的（因此也有相对限定名和完全限定名之分）。第3章已经说明函数、类和方法都有__qualname__特殊属性，引用的字符串即各自的限定名后缀。

下面的例子说明了如何通过__qualname__获得函数、类和方法的限定名后缀：

```
$ python3

>>> def f():
...     pass
...
>>> class C:
...     def g():
...         pass
...
>>> f.__qualname__
'f'
>>> C.__qualname__
'C'
>>> C.g.__qualname__
'C.g'
>>>
```

6-7. 导入模块的细节

（语言参考手册：5）
（标准库：内置类型、sys）

在本章的最后我们讨论模块导入过程的细节。本节的内容涉及sys模块中的一些属性，被总结在表6-1中。下面先简要介绍这些sys属性。（sys模块作为Python的运行时服务的核心模块将在第7章详细讨论。）

表6-1. 模块导入相关sys属性

属性	说明
sys.modules	一个字典，以记录所有已导入模块。
sys.builtin_module_names	一个字符串元组，以列出所有内置模块的模块名。
sys.stdlib_module_names	一个凝固集合，以列出标准库中所有模块的模块名。
sys.meta_path	一个列表，以列出所有元路径查找器。
sys.path_hooks	一个列表，其中的可调用对象被用于创建路径条目查找器。
sys.path	一个列表，以列出所有模块搜索路径。
sys.path_importer_cache	一个字典，作为路径条目查找器的缓存。

请通过下面的命令行和语句验证上述sys属性（显示内容太多时以省略号表示）：

```
      $ python3

      >>> import sys
      >>> sys.modules
      {'sys': <module 'sys' (built-in)>, 'builtins': <module 'builtins' (built-
in)>, ...}
      >>> sys.builtin_module_names
      ('_abc', '_ast', '_codecs', '_collections', '_functools', '_imp', '_io',
'_locale', '_operator', '_signal', '_sre', '_stat', '_string', '_symtable',
'_thread', '_tokenize', '_tracemalloc', '_warnings', '_weakref', 'atexit',
'builtins', 'errno', 'faulthandler', 'gc', 'itertools', 'marshal', 'posix',
'pwd', 'sys', 'time', 'xxsubtype')
      >>> sys.stdlib_module_names
      frozenset({'ipaddress', 'asynchat', ...})
      >>> sys.meta_path
      [<class '_frozen_importlib.BuiltinImporter'>, <class
'_frozen_importlib.FrozenImporter'>, <class
'_frozen_importlib_external.PathFinder'>]
      >>> sys.path_hooks
      [<class 'zipimport.zipimporter'>, <function
FileFinder.path_hook.<locals>.path_hook_for_FileFinder at 0x10297b910>]
      >>> sys.path
      ['', '/Library/Frameworks/Python.framework/Versions/3.11/lib/
python311.zip', '/Library/Frameworks/Python.framework/Versions/3.11/lib/
python3.11', '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
lib-dynload', '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
site-packages']
      >>> sys.path_importer_cache
      {'/Library/Frameworks/Python.framework/Versions/3.11/lib/python311.zip':
None, '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11':
FileFinder('/Library/Frameworks/Python.framework/Versions/3.11/lib/
python3.11'), ...}
      >>>
```

下面正式开始讨论模块导入过程。不论采用哪种语法，import语句被执行时都会在内部调用内置函数__import__()，该内置函数的语法为：

```
__import__(name, globals=None, locals=None, fromlist=(),
level=0)
```

其中各参数的含义是：

- name：模块的限定名。
- globals：一个代表全局名字空间的字典。
- locals：一个储存有全部本地变量的映射。
- fromlist：一个序列，为从该模块导入的标识符列表。
- level：一个非负整数，表明从当前模块/包的第几级祖先包开始搜索，以支持将相对限定名传入name。如果传入name的是完全限定名，则将0传入level。

__import__()会搜索指定的模块，如果找到则返回相应模块对象（若该模块对象尚不存在则创建它），然后import语句会继续完成被导入标识符的绑定。如果找不到指定的模块，则抛出ModuleNotFoundError异常。

这里给出一个直接使用__import__()的例子：

```
$ python3

>>> dir1 = __import__('dir1')
>>> dir1
<module 'dir1' from '/Users/wwy/dir1/__init__.py'>
>>> dir1.__file__
'/Users/wwy/dir1/__init__.py'
>>> dir1.__path__
['/Users/wwy/dir1']
>>> dir1.__cached__
'/Users/wwy/dir1/__pycache__/__init__.cpython-311.pyc'
>>>
```

但一般而言，我们不会直接使用__import__()。

接下来深入讨论__import__()的执行过程。这分为三个阶段：

- 缓存检查：检查模块缓存中是否已经有了所需模块对象，如果是则直接返回该对象，不进入后续阶段。
- 查找：通过“查找器（finders）”搜索指定的模块，并返回模块规格说明（一个ModuleSpec类型对象）。
- 加载：根据查找器返回的模块规格说明，通过“加载器（loaders）”创建模块对象。

需要强调的是，查找器和加载器不一定相互独立，如果一个对象同时实现了查找器和加载器的功能，则被称为“导入器（importers）”。

模块缓存是通过sys.modules引用的字典实现的，字典的键就是模块的限定名，而值则是相应模块对象。

导入一个模块会自动导入它所属的包，而导入一个包会自动导入它所属的父包，……，依此类推。不论import语句是否为这些模块对象绑定了标识符，sys.modules都会使这些模块对象的引用数不为0。举例来说，如果导入了foo.bar.baz模块，那么模块缓存中将增加三个键值对，键分别是“foo”、“foo.bar”和“foo.bar.baz”。

当需要导入一个新的模块时，首先会将该模块的相对限定名转换为完全限定名（如果直接给出完全限定名则跳过该操作），然后依次检查该完全限定名中涉及的包/模块是否已经存在于模块缓存中。如果否，则进入查找阶段，最后在加载阶段将所需的所有模块对象添加到模块缓存中。

虽然sys.module是可写的，但除非你知道你在做什么，否则不要手工修改它。如果删除了sys.modules中的键值对，那么相关模块对象即便存在也无法在模块缓存中找到，导致它被重新创建一次。同理，如果将sys.modules中的某个键值对的值改成None，则会强制使得每导入一次该模块（或其子模块）就要重新创建一个模块对象。最糟情况下，同一模块的模块对象可能会不一致。

当进入查找阶段后，会通过所谓的“元路径查找器（meta path finders）”进行查找，它们被存放在sys.meta_path引用的列表中。

从本节开头例子的结果中可以看出，Python解释器自带三个默认元路径查找器：

- _frozen_importlib.BuiltinImporter类型的对象：负责查找和加载内置模块。
- _frozen_importlib.FrozenImporter类型的对象：负责查找和加载冻结模块。
- _frozen_importlib_external.PathFinder类型的对象：负责查找模块。

其中前两个元路径查找器是导入器，只有PathFinder查找器是纯粹的查找器。

可以给sys.meta_path添加自定义的元路径查找器，它们必须是具有find_spec函数属性的对象，该函数的语法为：

```
find_spec(fullname, path, target=None)
```

其中fullname参数会被传入目标模块的完全限定名，path参数将被传入目标模块所属包的完全限定名（查找顶层模块时将传入None），而target参数仅在重新加载某模块时被传入当前对应目标模块的模块对象。该函数根据这些参数查找目标模块，如果找到则返回该目标模块的规格说明，否则返回None。

元路径查找的具体过程是这样的：针对需要加载的每一个模块，遍历sys.meta_path引用的列表，依次调用元路径查找器的find_spec属性，直到得到关于该模块的规格说明。如果所有元路径查找器的find_spec属性都返回了None，则说明当前模块无法被找到，应抛出ModuleNotFoundError异常。当得到了所有模块的规格说明，就进入加载阶段。

BuiltinImporter导入器和FrozenImporter导入器都能凭自身完成查找和导入操作。然而PathFinder查找器需要依赖于所谓的“路径条目查找器（path entry finders）”才能完成查找。路径查找器是动态生成的，而相关类储存在sys.path_hooks引用的列表中。

sys.path_hooks引用的列表默认包含两个对象，分别对应FileFinder类和zipimporter类。FileFinder的实例是默认的路径条目查找器，其实例化语法为：

```
class FileFinder(path, *loader_details)
```

其中path参数用于传入被查找的目录，loader_details参数则需要传入若干二元组，每个二元组都包含一个加载器和它能够识别的文件后缀名列表。即使没有给loader_details参数传入任何二元组，Python解释器默认的加载器也支持加载Python脚本及对应的字节码文件，以及动态链接库文件，也就是说支持.py、.pyc、.so、.dll、.pyd、……等后缀名。每个FileFinder对象都负责在一个目录中查找以指定模块名为文件名，具有特定后缀名的文件。FileFinder对象具有函数属性find_spec，以被用做查找器。

zipimporter类是由标准库中的zipimport模块定义的，其语法为：

```
class zipimporter(archivepath)
```

其中archivepath参数用于传入被查找的zip文件。每个zipimporter对象都负责在一个zip文件中查找模块，使得可以将ZIP归档文件等同于一个目录来处理。zipimporter对象也具有find_spec属性，但其本质是实现解压缩，然后自动为解压得到的目录创建一个FileFinder对象，并返回该FileFinder对象的find_spec属性的返回值。事实上，zipimporter对象还具有函数属性create_module和exec_module，因此准确的说是一个导入器。Python解释器启动时会自动导入zipimport模块。

我们可以通过给sys.path_hooks引用的列表添加类来引入自定义路径条目查找器，这些类至少需要具有find_spec函数属性。

实际上，PathFinder的find_spec是这样工作的：为sys.path引用的路径列表中的每个指向目录的条目创建一个路径条目查找器，然后以目标模块的完全限定名为参数调用这些路径条

目查找器的find_spec，并将得到的第一个模块规格说明作为自己的返回值。如果所有路径条目查找器的find_spec都返回None，则PathFinder的find_spec也返回None。而对于sys.path引用的路径列表中的每个指向ZIP归档文件的条目，将创建一个zipimporter对象作为导入器，进而直接完成导入，不需要返回模块规格说明。

本章前面已经提到了，sys.path引用的列表其实就是Python解释器的内置模块目录列表。该列表中总是包含一个空串以代表工作目录；此外还包含sys.prefix和sys.exec_prefix与sys.platlibdir（将在第7章讨论）共同指定的目录下的这些子目录和ZIP归档文件（这是针对Unix和类Unix操作系统的，Windows上需要去掉所有的“python3.11”）：

- python3.11：存放实现标准库中非内置模块的Python脚本。
- python3.11/lib-dynload：存放标准库运行时所需的共享库文件。
- python3.11/site-packages：存放第三方模块。
- python3.11.zip：对上述目录及其下的所有文件打包得到，作为它们的替代。

上述目录对所有用户都有效，而sys.path还可能包含存放登录用户专属第三方模块的目录，这会在第7章详细讨论。通过给sys.path引用的列表添加路径，可以增加PathFinder元路径查找器的搜索范围，这就是前面关于MyModules目录的例子中修改sys.path背后的原理。

但要注意，让sys.path包含工作目录其实是存在风险性的，可能导致在不同目录下执行同一个Python脚本的结果不同（该Python脚本导入了其他模块），甚至是一个可被攻击者利用的安全隐患。为解决这一问题，Python 3.11引入了-P选项和PYTHONSAFEPATH环境变量，如果在启动Python解释器时添加了该选项或设置了该环境变量，则sys.path默认不会包含代表工作目录的空串。

由于sys.path引用的列表是较少变化的，因此没有必要每次导入模块都动态创建路径条目查找器，而应该将它们缓存起来重复使用以提高执行速度。sys.path_importer_cache引用的字典就是路径条目查找器的缓存，而sys.path引用的列表中的每个路径都会成为该字典的一个键，而值就是对相应路径条目查找器的引用。但要注意，如果某个路径在文件系统中不存在，创建相应路径条目查找器的操作会失败，此时相应缓存条目将引用None。

进入加载阶段之后，会根据每个模块规格说明调用加载器，以完成对这些模块的加载。

这意味着先调用加载器的create_module属性以创建模块对象，然后更新sys.modules以缓存该模块对象，最后调用加载器的exec_module属性以初始化模块对象。加载器的返回值是该模块对象，这也就是__import__()的返回值。

我们可以自定义加载器，并在自定义路径条目查找器时使用它们。在初始化模块对象时，会将其__loader__实例属性设置为对被使用加载器的引用，不过如果使用的是Python解释器默认的加载器，则该属性将引用None。此外，模块对象的__spec__实例属性会在初始化时被设置为引用它的规格说明。

最后，如果你想对模块导入机制进行自定义，那么最好使用标准库中的importlib模块。事实上，importlib模块是模块导入机制的具体实现，因此其内提供的函数和类是最底层的。

举例来说，内置函数__import__()会在内部调用importlib.__import__()，而importlib.import_module()则是对importlib.__import__()的简化包装。这意味着如果我们通过调用importlib.import_module()导入模块的话，将跳过内置函数__import__()。

importlib模块还定义了很多抽象基类（将在第15章讨论）以帮助我们自定义查找器和加载器。

对importlib模块的详细讨论超出了本书的范围。

---

[1] "PEP 257 – Docstring Conventions". https://peps.python.org/pep-0257/.

[2] "PEP 420 – Implicit Namespace Packages". https://peps.python.org/pep-0420/.