

## 9. 类

类提供了把数据和功能绑定在一起的方法。创建新类时创建了对应的对象类型，从而能够创建该类型的新实例。实例具有能维持自身状态的属性，还具有能修改自身状态的方法（由其所属的类来定义）。

和其他编程语言相比，Python 的类只使用了很少的新语法和语义。Python 的类有点类似于 C++ 和 Modula-3 中类的结合体，而且支持面向对象编程（OOP）的所有标准特性：类的继承机制支持多个基类、派生的类能覆盖基类的方法、类的方法能调用基类中的同名方法。对象可包含任意数量和类型的数据。和模块一样，类也支持 Python 动态特性：在运行时创建，创建后还可以修改。

如果用 C++ 术语来描述的话，类成员（包括数据成员）通常为 *public*（例外的情况见下文[私有变量](#)），所有成员函数都为 *virtual*。与 Modula-3 中一样，没有用于从对象的方法中引用本对象成员的简写形式：方法函数在声明时，有一个显式的第一个参数代表本对象，该参数由方法调用隐式提供。与在 Smalltalk 中一样，Python 的类也是对象，这为导入和重命名提供了语义支持。与 C++ 和 Modula-3 不同，Python 的内置类型可以用作基类，供用户扩展。此外，与 C++ 一样，具有特殊语法的内置运算符（算术运算符、下标等）都可以为类实例重新定义。

由于缺乏关于类的公认术语，本章中偶尔会使用 Smalltalk 和 C++ 的术语。本章还会使用 Modula-3 的术语，Modula-3 的面向对象语义比 C++ 更接近 Python，但估计听说过这门语言的读者很少。

### 9.1. 名称和对象

对象之间相互独立，多个名称（甚至是多个作用域内的多个名称）可以绑定到同一对象。这在其他语言中通常被称为别名。Python 初学者通常不容易理解这个概念，处理数字、字符串、元组等不可变基本类型时，可以不必理会。但是，对于涉及可变对象（如列表、字典，以及大多数其他类型）的 Python 代码的语义，别名可能会产生意料之外的效果。这样做，通常是为了让程序受益，因为别名在某些方面就像指针。例如，传递对象的代价很小，因为实现只传递一个指针；如果函数修改了作为参数传递的对象，调用者就可以看到更改——无需像 Pascal 那样用两个不同的机制来传参。

### 9.2. Python 作用域和命名空间

在介绍类前，首先要介绍 Python 的作用域规则。类定义对命名空间有一些巧妙的技巧，了解作用域和命名空间的工作机制有利于加强对类的理解。并且，即便对于高级 Python 程序员，这方面的知识也很有用。

接下来，我们先了解一些定义。

*namespace*（命名空间）是从名称到对象的映射。现在，大多数命名空间都使用 Python 字典实现，但除非涉及到性能优化，我们一般不会关注这方面的事情，而且将来也可能会改变这种方式。命名空间的例子有：内置名称集合（包括 [abs\(\)](#) 函数以及内置异常的名称等）；一个模块的全局名称；一个函数调用中的局部名称。对象的属性集合也是命名空间的一种形式。关于命名空间的一个重要

知识点是，不同命名空间中的名称之间绝对没有关系；例如，两个不同的模块都可以定义 `maximize` 函数，且不会造成混淆。用户使用函数时必须要在函数名前面加上模块名。

点号之后的名称是 **属性**。例如，表达式 `z.real` 中，`real` 是对象 `z` 的属性。严格来说，对模块中名称的引用是属性引用：表达式 `modname.funcname` 中，`modname` 是模块对象，`funcname` 是模块的属性。模块属性和模块中定义的全局名称之间存在直接的映射：它们共享相同的命名空间！[\[1\]](#)

属性可以是只读的或者可写的。在后一种情况下，可以对属性进行赋值。模块属性是可写的：你可以写入 `modname.the_answer = 42`。也可以使用 `del` 语句删除可写属性。例如，`del modname.the_answer` 将从名为 `modname` 对象中移除属性 `the_answer`。

命名空间是在不同时刻创建的，且拥有不同的生命周期。内置名称的命名空间是在 Python 解释器启动时创建的，永远不会被删除。模块的全局命名空间在读取模块定义时创建；通常，模块的命名空间也会持续到解释器退出。从脚本文件读取或交互式读取的，由解释器顶层调用执行的语句是 `__main__` 模块调用的一部分，也拥有自己的全局命名空间。内置名称实际上也在模块里，即 `builtins`。

函数的局部命名空间在函数被调用时被创建，并在函数返回或抛出未在函数内被处理的异常时，被删除。（实际上，用“遗忘”来描述实际发生的情况会更好一些。）当然，每次递归调用都有自己的局部命名空间。

一个命名空间的 **作用域** 是 Python 代码中的一段文本区域，从这个区域可直接访问该命名空间。“可直接访问”的意思是，该文本区域内的名称在被非限定引用时，查找名称的范围，是包括该命名空间在内的。

作用域虽然是被静态确定的，但会被动态使用。执行期间的任何时刻，都会有 3 或 4 个“命名空间可直接访问”的嵌套作用域：

- 最内层作用域，包含局部名称，并首先在其中进行搜索
- 那些外层闭包函数的作用域，包含“非局部、非全局”的名称，从最靠内层的那个作用域开始，逐层向外搜索。
- 倒数第二层作用域，包含当前模块的全局名称
- 最外层（最后搜索）的作用域，是内置名称的命名空间

如果一个名称被声明为全局，则所有引用和赋值都将直接指向“倒数第二层作用域”，即包含模块的全局名称的作用域。要重新绑定在最内层作用域以外找到的变量，可以使用 `nonlocal` 语句；如果未使用 `nonlocal` 声明，这些变量将为只读（尝试写入这样的变量将在最内层作用域中创建一个 *新的* 局部变量，而使得同名的外部变量保持不变）。

通常，当前局部作用域将（按字面文本）引用当前函数的局部名称。在函数之外，局部作用域引用与全局作用域一致的命名空间：模块的命名空间。类定义在局部命名空间内再放置另一个命名空间。

划重点，作用域是按字面文本确定的：模块内定义的函数的全局作用域就是该模块的命名空间，无论该函数从什么地方或以什么别名被调用。另一方面，实际的名称搜索是在运行时动态完成的。但是，Python 正在朝着“编译时静态名称解析”的方向发展，因此不要过于依赖动态名称解析！（局部变量已经是被静态确定了。）

Python 有一个特殊规定。如果不存在生效的 `global` 或 `nonlocal` 语句，则对名称的赋值总是会进入最内层作用域。赋值不会复制数据，只是将名称绑定到对象。删除也是如此：语句 `del x` 从局部作用域引用的命名空间中移除对 `x` 的绑定。所有引入新名称的操作都是使用局部作用域：尤其是 `import` 语句和函数定义会在局部作用域中绑定模块或函数名称。

`global` 语句用于表明特定变量在全局作用域里，并应在全局作用域中重新绑定；`nonlocal` 语句表明特定变量在外层作用域中，并应在外层作用域中重新绑定。

### 9.2.1. 作用域和命名空间示例

下例演示了如何引用不同作用域和名称空间，以及 `global` 和 `nonlocal` 对变量绑定的影响：

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

示例代码的输出是：

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

注意，**局部** 赋值（这是默认状态）不会改变 `scope_test` 对 `spam` 的绑定。`nonlocal` 赋值会改变 `scope_test` 对 `spam` 的绑定，而 `global` 赋值会改变模块层级的绑定。

而且，`global` 赋值前没有 `spam` 的绑定。

## 9.3. 初探类

类引入了一点新语法，三种新的对象类型和一些新语义。

### 9.3.1. 类定义语法

最简单的类定义形式如下：

```
class ClassName:
    <语句-1>
    .
    .
    <语句-N>
```

与函数定义 (`def` 语句) 一样，类定义必须先执行才能生效。把类定义放在 `if` 语句的分支里或函数内部试试。

在实践中，类定义内的语句通常都是函数定义，但也可以是其他语句。这部分内容稍后再讨论。类里的函数定义一般是特殊的参数列表，这是由方法调用的约定规范所指明的 --- 同样，稍后再解释。

当进入类定义时，将创建一个新的命名空间，并将其用作局部作用域 --- 因此，所有对局部变量的赋值都是在这个新命名空间之内。特别的，函数定义会绑定到这里的新函数名称。

当 (从结尾处) 正常离开类定义时，将创建一个 **类对象**。这基本上是一个围绕类定义所创建的命名空间的包装器；我们将在下一节中了解有关类对象的更多信息。原始的 (在进入类定义之前有效的) 作用域将重新生效，类对象将在这里与类定义头所给出的类名称进行绑定 (在这个示例中为 `ClassName`)。

### 9.3.2. Class 对象

类对象支持两种操作：属性引用和实例化。

**属性引用** 使用 Python 中所有属性引用所使用的标准语法: `obj.name`。有效的属性名称是类对象被创建时存在于类命名空间中的所有名称。因此，如果类定义是这样的：

```
class MyClass:
    """一个简单的示例类"""
    i = 12345

    def f(self):
        return 'hello world'
```

那么 `MyClass.i` 和 `MyClass.f` 就是有效的属性引用，将分别返回一个整数和一个函数对象。类属性也可以被赋值，因此可以通过赋值来改变 `MyClass.i` 的值。 `__doc__` 也是一个有效的属性，将返回所属类的文档字符串: `"A simple example class"`。

类的 **实例化** 使用函数表示法。可以把类对象视为是返回该类的一个新实例的不带参数的函数。举例来说 (假设使用上述的类)：

```
x = MyClass()
```

创建类的新 **实例** 并将此对象分配给局部变量 `x`。

实例化操作 (“调用”类对象) 会创建一个空对象。许多类都希望创建的对象实例是根据特定初始状态定制的。因此一个类可能会定义名为 `__init__()` 的特殊方法，就像这样：

```
def __init__(self):
    self.data = []
```

当一个类定义了 `__init__()` 方法时，类的实例化会自动为新创建的类实例唤起 `__init__()`。因此在这个例子中，可以通过以下语句获得一个已初始化的新实例：

```
x = MyClass()
```

当然，`__init__()` 方法还有一些参数用于实现更高的灵活性。在这种情况下，提供给类实例化运算符的参数将被传递给 `__init__()`。例如，

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

### 9.3.3. 实例对象

现在我们能用实例对象做什么？实例对象所能理解的唯一操作是属性引用。有两种有效的属性名称：数据属性和方法。

数据属性相当于 Smalltalk 中的 "实例变量" 和 C++ 中的 "数据成员"。数据属性无需声明；与局部变量一样，它们在首次赋值时就会出现。例如，如果 `x` 是上面创建的 `MyClass` 的实例，那么下面的代码将打印值 `16`，而不会留下任何痕迹：

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

另一种实例属性引用称为 方法。方法是“从属于”对象的函数。

实例对象的有效方法名称依赖于其所属的类。根据定义，一个类中所有是函数对象的属性都是定义了其实例的相应方法。因此在我们的示例中，`x.f` 是有效的方法引用，因为 `MyClass.f` 是一个函数，而 `x.i` 不是方法，因为 `MyClass.i` 不是函数。但是 `x.f` 与 `MyClass.f` 并不是一回事 --- 它是一个 方法对象，不是函数对象。

### 9.3.4. 方法对象

通常，方法在绑定后立即被调用：

```
x.f()
```

如果 `x = MyClass()`，就像上面这样，则将返回字符串 `'hello world'`。不过，方法并非必须立即被调用：`x.f` 是一个方法对象，并可被存储起来并在以后再调用。例如：

```
xf = x.f
while True:
    print(xf())
```

将持续打印 `hello world`，直到结束。

当一个方法被调用时究竟会发生什么？你可能已经注意到尽管 `f()` 的函数定义指定了一个参数，但上面调用 `x.f()` 时却没有带参数。这个参数发生了什么事？当一个需要参数的函数在不附带任何参数的情况下被调用时 Python 肯定会引发异常 --- 即使参数实际上没有被使用...

实际上，你可能已经猜到了答案：方法的特殊之处就在于实例对象会作为函数的第一个参数被传入。在我们的示例中，调用 `x.f()` 其实就相当于 `MyClass.f(x)`。总之，调用一个具有  $n$  个参数的方法就相当于调用再多一个参数的对应函数，这个参数值为方法所属实例对象，位置在其他参数之前。

总而言之，方法的运作方式如下。当一个实例的非数据属性被引用时，将搜索该实例所属的类。如果名称表示一个属于函数对象的有效类属性，则指向实例对象和函数对象的引用将被打包为一个方法对象。当传入一个参数列表调用该方法对象时，将基于实例对象和参数列表构造一个新的参数列表，并传入这个新参数列表调用相应的函数对象。

### 9.3.5. 类和实例变量

一般来说，实例变量用于每个实例的唯一数据，而类变量用于类的所有实例共享的属性和方法：

```
class Dog:

    kind = 'canine'          # 类变量被所有实例所共享

    def __init__(self, name):
        self.name = name    # 实例变量为每个实例所独有

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                # 被所有的 Dog 实例所共享
'canine'
>>> e.kind                # 被所有的 Dog 实例所共享
'canine'
>>> d.name                # 为 d 所独有
'Fido'
>>> e.name                # 为 e 所独有
'Buddy'
```

正如 [名称和对象](#) 中已讨论过的，共享数据可能在涉及 [mutable](#) 对象例如列表和字典的时候导致令人惊讶的结果。例如以下代码中的 `tricks` 列表不应该被用作类变量，因为所有的 `Dog` 实例将只共享一个单独的列表：

```
class Dog:

    tricks = []             # 类变量的错误用法

    def __init__(self, name):
        self.name = name
```



```

def add_trick(self, trick):
    self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks          # 非预期地被所有的 Dog 实例所共享
['roll over', 'play dead']

```

正确的类设计应该使用实例变量:

```

class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # 为每个 Dog 实例新建一个空列表

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']

```

## 9.4. 补充说明

如果同样的属性名称同时出现在实例和类中，则属性查找会优先选择实例:

```

>>> class Warehouse:
...     purpose = 'storage'
...     region = 'west'
...
>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east

```

数据属性可以被方法以及一个对象的普通用户（“客户端”）所引用。换句话说，类不能用于实现纯抽象数据类型。实际上，在 Python 中没有任何东西能强制隐藏数据 --- 它是完全基于约定的。

（而在另一方面，用 C 语言编写的 Python 实现则可以完全隐藏实现细节，并在必要时控制对象的访问；此特性可以通过用 C 编写 Python 扩展来使用。）

客户端应当谨慎地使用数据属性 --- 客户端可能通过直接操作数据属性的方式破坏由方法所维护的固定变量。请注意客户端可以向一个实例对象添加他们自己的数据属性而不会影响方法的可用性，只要保证避免名称冲突 --- 再次提醒，在此使用命名约定可以省去许多令人头痛的麻烦。

在方法内部引用数据属性（或其他方法！）并没有简便方式。我发现这实际上提升了方法的可读性：当浏览一个方法代码时，不会存在混淆局部变量和实例变量的机会。

方法的第一个参数常常被命名为 `self`。这也不过就是一个约定：`self` 这一名称在 Python 中绝对没有特殊含义。但是要注意，不遵循此约定会使得你的代码对其他 Python 程序员来说缺乏可读性，而且也可以想像一个 *类浏览器* 程序的编写可能会依赖于这样的约定。

任何一个作为类属性的函数都为该类的实例定义了一个相应方法。函数定义的文本并非必须包含于类定义之内：将一个函数对象赋值给一个局部变量也是可以的。例如：

```
# 在类之外定义的函数
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

h = g
```

现在 `f`、`g` 和 `h` 都 `C` 类的指向函数对象的属性，因此它们都是 `C` 实例的方法 --- 其中 `h` 与 `g` 完全等价。但请注意这种做法通常只会使程序的阅读者感到迷惑。

方法可以通过使用 `self` 参数的方法属性调用其他方法：

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

方法可以通过与普通函数相同的方式引用全局名称。与方法相关联的全局作用域就是包含该方法的定义语句的模块。（类永远不会被用作全局作用域。）尽管一个人很少会有好的理由在方法中使用全局作用域中的数据，全局作用域依然存在许多合理的使用场景：举个例子，导入到全局作用域的函数和模块可以被方法所使用，定义在全局作用域中的函数和类也一样。通常，包含该方法的类本身就定义在全局作用域中，而在下一节中我们将会发现，为何有些时候方法需要引用其所属类。

每个值都是一个对象，因此具有 *类*（也称为 *类型*），并存储为 `object.__class__`。

## 9.5. 继承

当然，如果不支持继承，语言特性就不值得称为“类”。派生类定义的语法如下所示：

```
class DerivedClassName(BaseClassName):
    <语句-1>
```



```
.  
.   
.   
<语句-N>
```

名称 `BaseClassName` 必须定义于可从包含所派生的类的定义的作用域访问的命名空间中。作为基类名称的替代，也允许使用其他任意表达式。例如，当基类定义在另一个模块中时，这就会很有用处：

```
class DerivedClassName(modname.BaseClassName):
```

派生类定义的执行过程与基类相同。当构造类对象时，基类会被记住。此信息将被用来解析属性引用：如果请求的属性在类中找不到，搜索将转往基类中进行查找。如果基类本身也派生自其他某个类，则此规则将被递归地应用。

派生类的实例化没有任何特殊之处：`DerivedClassName()` 会创建该类的一个新实例。方法引用将按以下方式解析：搜索相应的类属性，如有必要将按基类继承链逐步向下查找，如果产生了一个函数对象则方法引用就生效。

派生类可能会重写其基类的方法。因为方法在调用同一对象的其他方法时没有特殊权限，所以基类方法在尝试调用调用同一基类中定义的另一方法时，可能实际上调用是该基类的派生类中定义的方法。（对 C++ 程序员的提示：Python 中所有的方法实际上都是 `virtual` 方法。）

在派生类中的重写方法实际上可能想要扩展而非简单地替换同名的基类方法。有一种方式可以简单地直接调用基类方法：即调用 `BaseClassName.methodname(self, arguments)`。有时这对客户端来说也是有用的。（请注意仅当此基类可在全局作用域中以 `BaseClassName` 的名称被访问时方可使用此方式。）

Python 有两个内置函数可被用于继承机制：

- 使用 `isinstance()` 来检查一个实例的类型：`isinstance(obj, int)` 仅会在 `obj.__class__` 为 `int` 或某个派生自 `int` 的类时为 `True`。
- 使用 `issubclass()` 来检查类的继承关系：`issubclass(bool, int)` 为 `True`，因为 `bool` 是 `int` 的子类。但是，`issubclass(float, int)` 为 `False`，因为 `float` 不是 `int` 的子类。

### 9.5.1. 多重继承

Python 也支持一种多重继承。带有多个基类的类定义语句如下所示：

```
class DerivedClassName(Base1, Base2, Base3):  
    <语句-1>  
    .  
    .  
    .  
    <语句-N>
```

对于多数目的来说，在最简单的情况下，你可以认为搜索从父类所继承属性的操作是深度优先、从左到右的，当层次结构存在重叠时不会在同一个类中搜索两次。因此，如果某个属性在

DerivedClassName 中找不到，就会在 Base1 中搜索它，然后（递归地）在 Base1 的基类中搜索，如果在那里也找不到，就将在 Base2 中搜索，依此类推。

真实情况比这个更复杂一些；方法解析顺序会动态改变以支持对 [super\(\)](#) 的协同调用。这种方式在某些其他多重继承型语言中被称为后续方法调用，它比单继承型语言中的 super 调用更强大。

动态调整顺序是有必要的，因为所有多重继承的情况都会显示出一个或更多的菱形关联（即至少有一个上级类可通过多条路径被最底层的类所访问）。例如，所有类都是继承自 [object](#)，因此任何多重继承的情况都提供了一条以上的路径可以通向 [object](#)。为了确保基类不会被访问一次以上，动态算法会用一种特殊方式将搜索顺序线性化，保留每个类所指定的从左至右的顺序，只调用每个上级类一次，并且保持单调性（即一个类可以被子类化而不影响其父类的优先顺序）。总而言之，这些特性使得设计具有多重继承的可靠且可扩展的类成为可能。要了解更多细节，请参阅 [Python 2.3 方法解析顺序](#)。

## 9.6. 私有变量

那种仅限从一个对象内部访问的“私有”实例变量在 Python 中并不存在。但是，大多数 Python 代码都遵循这样一个约定：带有一个下划线的名称（例如 `_spam`）应该被当作是 API 的非公有部分（无论它是函数、方法或是数据成员）。这应当被视为一个实现细节，可能不经通知即加以改变。

由于存在对于类私有成员的有效使用场景（例如避免名称与子类所定义的名称相冲突），因此存在对此种机制的有限支持，称为 **名称改写**。任何形式为 `__spam` 的标识符（至少带有两个前缀下划线，至多一个后缀下划线）的文本将被替换为 `_classname__spam`，其中 `classname` 为去除了前缀下划线的当前类名称。这种改写不考虑标识符的句法位置，只要它出现在类定义内部就会进行。

**参见：** [私有名称调整规范说明](#) 了解相关详情和特例。

名称改写有助于让子类重写方法而不破坏类内方法调用。例如：

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update    # 原始 update() 方法的私有副本

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # 为 update() 提供了新的签名
        # 但不会破坏 __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

上面的示例即使在 MappingSubclass 引入了一个 `__update` 标识符的情况下也不会出错，因为它会在 Mapping 类中被替换为 `_Mapping__update` 而在 MappingSubclass 类中被替换为

`__MappingSubclass__update`。

请注意，改写规则的设计主要是为了避免意外冲突；访问或修改被视为私有的变量仍然是可能的。这在特殊情况下甚至会很有用，例如在调试器中。

请注意传递给 `exec()` 或 `eval()` 的代码不会将唤起类的类名视作当前类；这类似于 `global` 语句的效果，因此这种效果仅限于同时经过字节码编译的代码。同样的限制也适用于 `getattr()`，`setattr()` 和 `delattr()`，以及对于 `__dict__` 的直接引用。

## 9.7. 杂项说明

有时具有类似于 Pascal "record" 或 C "struct" 的数据类型是很有用的，将一些带名称的数据项捆绑在一起。实现这一目标的理想方式是使用 [dataclasses](#)：

```
from dataclasses import dataclass
```

```
@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

一段期望使用特定抽象数据类型的 Python 代码通常可以通过传入一个模拟了该数据类型的方法的类作为替代。例如，如果你有一个基于文件对象来格式化某些数据的函数，你可以定义一个带有 [read\(\)](#) 和 [readline\(\)](#) 方法以便从字典串缓冲区获取数据的类，并将其作为参数传入。

[实例方法对象](#) 也具有属性：`m.__self__` 就是带有 `m()` 方法的实例对象，而 `m.__func__` 就是该方法所对应的 [函数对象](#)。

## 9.8. 迭代器

到目前为止，您可能已经注意到大多数容器对象都可以使用 [for](#) 语句：

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

这种访问风格清晰、简洁又方便。迭代器的使用非常普遍并使得 Python 成为一个统一的整体。在幕后，`for` 语句会在容器对象上调用 `iter()`。该函数返回一个定义了 `__next__()` 方法的迭代器对象，此方法将逐一访问容器中的元素。当元素用尽时，`__next__()` 将引发 `StopIteration` 异常来通知终止 `for` 循环。你可以使用 `next()` 内置函数来调用 `__next__()` 方法；这个例子显示了它的运作方式：

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

了解了迭代器协议背后的机制后，就可以轻松地为你的类添加迭代器行为了。定义 `__iter__()` 方法用于返回一个带有 `__next__()` 方法的对象。如果类已定义了 `__next__()`，那么 `__iter__()` 可以简单地返回 `self`：

```
class Reverse:
    """ 对一个序列执行反向循环的迭代器。 """
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

## 9.9. 生成器

[生成器](#) 是一个用于创建迭代器的简单而强大的工具。它们的写法类似于标准的函数，但当它们要返回数据时会使用 [yield](#) 语句。每次在生成器上调用 [next\(\)](#) 时，它会从上次离开的位置恢复执行（它会记住上次执行语句时的所有数据值）。一个显示如何非常容易地创建生成器的示例如下：

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

可以用生成器来完成的任何功能同样可以通用前一节所描述的基于类的迭代器来完成。但生成器的写法更为紧凑，因为它会自动创建 [\\_\\_iter\\_\\_\(\)](#) 和 [\\_\\_next\\_\\_\(\)](#) 方法。

另一个关键特性在于局部变量和执行状态会在每次调用之间自动保存。这使得该函数相比使用 `self.index` 和 `self.data` 这种实例变量的方式更易编写且更为清晰。

除了会自动创建方法和保存程序状态，当生成器终结时，它们还会自动引发 [StopIteration](#)。这些特性结合在一起，使得创建迭代器能与编写常规函数一样容易。

## 9.10. 生成器表达式

某些简单的生成器可以写成简洁的表达式代码，所用语法类似列表推导式，但外层为圆括号而非方括号。这种表达式被设计用于生成器将立即被外层函数所使用的情況。生成器表达式相比完整的生成器更紧凑但较不灵活，相比等效的列表推导式则更为节省内存。

示例:

```
>>> sum(i*i for i in range(10))           # 平方和
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))   # 点乘
260

>>> unique_words = set(word for line in page for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

**备注**

- [1] 存在一个例外。模块对象有一个秘密的只读属性名为 `__dict__`，它返回用于实现模块命名空间的字典；名称 `__dict__` 是一个属性但不是全局名称。显然，使用个将违反命名空间实现的抽象，应当仅限用于事后调试器之类的情况。