

source/python_tricks/1_简介.md

第一章 简介

1.1 什么是Python技巧

Python技巧：一段作为教学工具的简短Python代码。要么通过简单的示例讲授Python某个方面的特性，要么作为激励的示例，使你能够更进一步。

Python技巧一开始是我在Twitter上分享的一系列代码截图，整个过程持续了一周。反响之强烈令我震惊，分享和转发连续了好几天。

越来越多的开发人员开始要求我提供“获得整个系列”的方法。实际上，我只列出了其中的一些技巧，涵盖了与Python有关的很多主题。他们背后并没有总体的规划。他们只是一个有趣的Twitter小实验。

但是从这些询问中，我感觉到简短代码示例值得作为教学工具进行探索。最终，我开始创建更多的Python技巧，并在电子邮件中分享了它们。几天之内，几百个Python开发人员就加入了，我对这种回应感到很震惊。

在接下来的一段时间中，我不断接触到Python开发人员。他们很感谢我，我让他们正在努力理解的知识点变得很容易学习。听到这个反馈真是太棒了。我以为这些Python技巧只是代码截图，但是许多开发人员从中获益良多。

在那时，我决定加倍投入Python技巧实验并将其扩展为一个大约30封电子邮件的系列。这些仍然是用标题和代码截图的形式，但我很快意识到了这种格式的局限性。就在这个时候，一个盲人Python程序员给我发了封电子邮件，他对这些Python技巧使用图片进行发布感到很失望，因为他的屏幕阅读器无法阅读图片。

显然，我需要在这个项目上投入更多的时间，以使其更具吸引力，并被更多的受众所使用。因此，我开始用纯文本重写整个Python技巧电子邮件系列，并使用基于HTML的语法进行高亮。*Python技巧*的新版本很好地运转了一段时间。根据我收到的回复，开发人员似乎很高兴，他们终于可以复制粘贴示例并进行把玩了。

随着越来越多的开发人员加入了该电子邮件系列，我开始注意到收到的答复和问题中的共性问题。某些技巧本身作为激励示例很好地发挥了作用。但是，对于较复杂的示例，没有人来指导读者或给他们提供额外的资源来加深理解。

这是另一个需要改进的地方。dbader.org 的使命是帮助Python开发人员变得更加出色——这显然是一个接近该目标的机会。

我决定从电子邮件课程中选出最好，最有价值的Python技巧，然后开始围绕它们编写一本新的Python书籍：

一本以简短易懂的示例讲授语言中最酷的知识点的书。一本像自助餐一样的书，讲述Python中那些很棒的特性，并保持较高的激励作用。一本可以指导并帮助您加深对Python理解的书。

这本书对我来说确实是爱的劳动，也是一项巨大的实验。希望您会喜欢它，并在阅读的过程中学习Python相关的知识！——Dan Bader

source/python_tricks/2_整洁代码的模式.md

第二章 整洁代码的模式

2.1 使用断言

有时候，一些真正有用的语言特性往往没有得到应有的重视。Python的 assert 语句就是如此

在这一章我将会介绍如何使用Python的断言。你会学到如何使用断言自动捕获代码中的错误，这会增加代码的可读性，会让你的代码更容易调试。

在这个点上，你可能会疑惑，“什么是断言，断言有什么好处？”

Python的断言可以通过测试一个条件辅助调试。如果断言的条件是true，什么都不会发生，代码继续正常执行。但是如果条件是false，会抛出一个 AssertionError 的异常，这个异常可以添加一个可选的错误信息

Python中断言的一个例子

在这个简单的例子中，你可以看到断言的作用。我尽量使这个例子和真实世界中你可能遇到的问题更相似

假设你在使用Python搭建一个在线商店。你写下了下面的apply_discount函数为系统添加一个打折优惠的功能。

```
def apply_discount(product, discount):
    price = int(product['price'] * (1.0 - discount))
    assert 0 <= price <= product['price']
    return price
```

上面函数中的assert语句会保证在任何情况下，经过这个函数计算后的打折后的价格都不会比0更低，不会比初始价格更高。

让我们确认一下，在我们调用这个函数计算一个折扣的时候，这个逻辑确实按照我们设想的方式工作。在这个例子中，我们商店中的商品会使用字典进行表示。这很可能和你在真实应用中的做法不同，但是对于演示断言，这已经足够了。我们先创建一个商品当做例子，这个商品是一双价值\$149的漂亮鞋子

```
shoes = {'name': 'Fancy Shoes', 'price': 14900}
```

顺便说一下，你有没有注意到我使用整数来表示以分为单位的价格，从而避免了找零时的四舍五入问题？这是一个好主意，但是我不这么认为。现在，如果给予25%的折扣，销售价格就会是\$111.75

```
apply_discount(shoes, 0.25)
>>> 11175
```

一切正常。现在我们试试一些无效的折扣，比如200%的折扣，将会让我们倒付钱给消费者 >>> apply_discount(shoes, 2.0)

```
Traceback (most recent call last):
File "<input>", line 1, in <module>
    apply_discount(prod, 2.0)
File "<input>", line 4, in apply_discount
    assert 0 <= price <= product['price']
AssertionError
```

你看，当我们尝试打一个无效的折扣的时候，我们的程序抛出了一个 `AssertionError`。之所以会这样是因为200%的折扣违反了我们在 `apply_discount` 函数中的断言条件 你还能看到异常错误栈指出了包含失败断言的哪一行代码，如果你(或者你们团队的其它开发者)在测试在线商店的时候遇到了这样的一个错误，通过错误栈会很容易找到具体发生了什么

这会在很大程度上加快调试的速度，并且会让你的代码更容易维护。这就是断言的力量

为什么不直接使用一个常规的异常

你可能会疑惑为什么在上面的例子中我不直接使用 `if` 语句抛出一个异常。

正确使用断言可以通知开发者那些程序中不可恢复的错误。断言不是为了提示预期中的错误，对于像文件没找到这种预期中的错误，用户可以采取纠正措施，或者直接重试

断言意味着程序的内部自检。断言会声明一些不可能发生的条件，如果这些条件没有被兜住，意味着代码中存在bug

如果你的代码没有任何bug，这些断言条件永远不会发生。但是一旦发生，程序就会崩溃，此时会抛出一个断言错误，告诉你哪个“不可能”的条件被触发了。这会让定位和修复bug变得简单。我喜欢任何让生活变得简单的事情--难道你不是吗？

到目前为止，牢记Python的断言是一个调试的辅助工具，不是一套处理运行时错误的机制。使用断言是为了让开发者能够更快发现可能导致bug的根源。如果程序没有bug，断言错误永远不应该被主动抛出

让我们看一下可以用断言做的其它事情，后面我将会列出2个实践中常见的陷阱

Python的断语法

在使用一个Python的特性之前，最好学习一下它是怎样实现的。我们快速看看Python文档中断言的语法

```
assert_stmt ::= "assert" expression1 ["," expression2]
```

在这个例子中，`expression1`是我们测试的条件，可选的`expression2`是一个断言失败后显示的错误信息。在执行时，Python解释器吧每个断言转换成大致如下的语句：

```
if __debug__:
    if not expression1:
        raise AssertionError(expression2)
```

这段代码里面有两件很有趣的事情：

在断言条件检查之前，对全局变量`__debug__`进行了额外的检查。`__debug__`是一个内建的bool值，正常情况下这个值是`true`，如果需要优化，这个值是`false`在后面常见陷阱部分，我们会对这个问题进行更深入的讨论

你还可以使用`expression2`传入一个可选的错误信息，它会在 `AssertionError` 的错误栈中显示。这可以简化调试，比如下面的例子

```
if cond == 'x':
    ... do_x()
elif cond == 'y':
    ... do_y()
else:
    ... assert False,
    ... 'This should never happen, but it does '
    ... 'occasionally. We are currently trying to '
    ... 'figure out why. Email dbader if you '
    ... 'encounter this in the wild. Thanks!')
```

这难看吗？这确实是一种难看的写法，但在你遇到难以复现的bug的时候，这确实是一种有用的方法。

Python中断言的常见陷阱

在继续之前，我还想指出两个和Python中断言相关的重要警告。第一个和引入安全问题和bug有关，第二个和很容易写出无用断言的奇怪语法有关。

这听起来很恐怖，所以你最好至少遵守下面的两个警告

警告1：不要用断言进行数据验证

在使用Python断言时最大的警告：断言可以通过`-O`和`--O`命令行开关进行全局禁用，CPython中的`PYTHONOPTIMIZE`环境变量也可以禁用断言

这会将所有的断言语句变成空操作：断言会在编译时被排除，并且永远不会被评估，这意味着没有任何断言会被执行

这是一个设计决策，在很多其它编程语言中也有类似的决策。这种行为的副作用就是，使用断言对输入数据进行快速验证是极度危险的。

如果你的代码使用断言检查函数的参数是否包含一个“错误”的或者不在预期之内的值，这可能会适得其反，导致一些bug或者安全漏洞。

让我们用一个简单例子说明这个问题。假设你在使用Python搭建一个在线商城。在你应用的某个地方有一个函数根据用户的请求删除产品。

因为你刚刚学过断言，所以你很渴望在代码中使用断言，你写下了下面的代码。

```
def delete_product(prod_id, user):
    assert user.is_admin(), 'Must be admin'
    assert store.has_product(prod_id), 'Unknown product'
    store.get_product(prod_id).delete()
```

仔细看看这个`delete_product`函数，如果断言被禁用，将会发生什么？

在这个三行代码的函数中，有两个很严重的问题，它们都是因为不正确地使用断言导致的。

- 使用断言检查管理员权限是很危险的。如果在Python解释器中，断言被禁用，这将变成一个空操作。所以任何人都可以删除产品。权限检查的代码甚至根本没有运行。这很有可能引入安全问题，并且为攻击者摧毁或者严重损坏我们在线商城的数据打开大门。这不是一个好的用法。
- `has_product()`检查在断言禁用的时候会直接跳过。这意味着`get_product()`可以通过无效的产品id进行调用--这会导致更严重的bug。在最坏的情况下，这很可能是一个导致服务拒绝攻击的途径。比如，可能商城崩溃了，可能有人尝试删除了一个未知的商品，一个攻击者可能会通过无效的删除请求进行轰炸，进而导致服务中断。

我们怎么避免这些问题呢？答案就是永远不要使用断言进行数据验证。我们可以使用常规的if语句进行验证，在必要的时候抛出验证异常，比如：

```
def delete_product(product_id, user):
    if not user.is_admin():
        raise AuthError('Must be admin to delete')
    if not store.has_product(product_id):
        raise ValueError('Unknown product id')
    store.get_product(product_id).delete()
```

这个更新后的例子还有一个好处：它现在抛出像`ValueError`或者`AuthError`(我们必须自己定义)这类语义上正确地异常，而不是抛出一个不确定的断言异常。

警告2: 永不失败的断言

偶然写出永远为真的Python断言是一件相当容易的事情。我在过去就尝到过这样的苦果。

TODO 补充一句

当你将元组做为第一个参数传给断言的时候，断言总是为真，从不失败。下面的例子中的断言永远不会失败。

```
assert(1 == 2, 'This should fail')
```

这和Python中不为空的元祖总是为真有关。如果你传一个元祖给一个断言，会导致断言条件永远是真--进而导致上面的断言语句永远不会失败，永远不会发出一个异常。

相对来说很容易因为直觉行为写出这种多行的糟糕断言。比如，我很快乐的在我的一个测试套件中写了一堆无用的测试用例，这些用例给人以一种错误的安全感。想象一下在你的一个单元测试中有如下的断言。

```
assert (
    counter == 10,
    'It should have counted all the items'
)
```

在第一次检测的时候，这个测试案例看起来没有任何问题。然而它永远不会捕获错误的结果：无论变量counter的值为何，断言总是为真。为什么会这样呢？因为它在对一个值为真的元组进行断言。

就像我说过的，这种做法很容易砸到自己的脚。一个很好的防止这种奇怪预发引起麻烦的对策是使用[code linter](#) Python3的新版本会对这种模糊的断言给出警告。

顺便说一句，这就是为什么你应该始终对单元测试用例进行快速烟雾测试的原因。确保它们确实可以失败然后继续写下一个。

Python断言--总结

除了这些经过，我坚信Python的断言是一个没有被Python开发者充分使用的调试工具。

懂得断言怎么工作，在什么时候使用断言可以帮助你写出更容易维护的和更容易调试的代码。

这是一项很棒的学习技能，它可让你的Python知识更进一步，使你成为更全面的Pythonista。我知道它节省了我数小时的调试工作。

重点

- Python的assert语句是一种调试辅助工具，可以使用内部自检的方式测试一个条件。
- 断言仅应用于帮助开发人员识别错误。它们不是用于处理运行时错误的机制。
- 可以通过解释器设置在全局禁用断言。

2.2 逗号放置

当你在Python中的列表，字典或集合中增删元素的时候，有一个有用的建议：以逗号作为所有行的结尾。

不确定我在说什么？让我来给出一个例子.想象一下你在代码里面有这样一个名字的列表.

```
>>> names = ['Alice', 'Bob', 'Dilbert']
```

当你修改这个列表的时候，通过git的diff命令很难看到具体修改了什么。很多的源码管理系统是基于行的，很难凸显在一行中的多出修改。

一个快速的解决方法是采用一种更新的代码风格，将列表，字典，集合像下面这样展开到多个行。

```
>>> names = [
... 'Alice',
... 'Bob',
... 'Dilbert'
... ]
```

这样每一行就只有一个元素，添加，修改，删除一个元素在代码管理系统中很容易通过diff进行显示。这是一个简单的改动，但是帮我避免了愚蠢的错误。这也让我团队的同事在review我的代码的时候更加轻松。

现在还有两个涉及到编辑的情况会引起困惑。当你在列表的末尾添加一个元素，或者你删除了最后一个元素，你不得不修改逗号的位置以保持统一的代码风格。

比如你想要在列表添加另一个名字（Jane）。如果你添加了Jane，你就需要修改在Dilbert这一行所在的逗号来避免错误。

```
>>> names = [
... 'Alice',
... 'Bob',
... 'Dilbert' # <- Missing comma!
... 'Jane'
]
```

当你检查列表的内容时，你可能会感到惊奇：

```
>>> names
['Alice', 'Bob', 'DilbertJane']
```

你看，Python把字符串Dilbert和Jane合并成了一个新的字符串 DilbertJane。这种所谓的“字符串字面串联”是有意设计的，在文档里也有说明。这是一种音符难以捕获的bug到代码中，搬起石头砸自己的脚的匪夷所思的方式。

多个相邻的字符或者字节（使用空格分割）是允许的，他们可能使用不同的引号，它们的意义和它们的连接体是一样的。

字符串字面连接在某些情况下还是一个有用的特性。比如，你可以用它减少分割长字符串到多个行时需要的反斜杆。

```
my_str = ('This is a super long string constant '
          'spread out across multiple lines. '
          'And look, no backslash characters needed!')
```

另一方面，我们刚刚看到相同的功能如何快速导致一种问题。现在我们怎么解决这个问题？

在*Dilbert*行后面添加逗号会防止两个字符拼接成一个。

```
>>> names = [
... 'Alice',
... 'Bob',
... 'Dilbert',
... 'Jane'
]
```

但是现在我们绕了一圈又回到了原点。为了添加一个新的名字到列表，我必须修改两行。这导致在 `git diff` 时很难看到什么被修改了。是有人添加了一个新的名字，还是有人修改了*Dilbert*的名字？

幸运的是，Python的语法允许一些回旋余量来一劳永逸地解决这个逗号放置的问题。你只需要训练自己采用一种一开始就避免它的代码风格。让我具体来解释。

在Python中，你可以在列表，字典，集合中为每一个元素（包含最后一个元素）后面都添加一个逗号，你只需要记得让每一行都使用逗号结尾，这样你就可以避免对是否放置逗号的判断。

最终的例子如下：

```
>>> names = [
... 'Alice',
... 'Bob',
... 'Dilbert',
... ]
```

你有没有注意到*Dilbert*这一行最后的逗号，这样可以在新增或者删除元素的时候不必在更新逗号的位置。这让你的代码保持一致，代码管理系统的diff操作更加清晰明了，代码评审的同事更加开心。有时候，魔法藏在细节中，不是吗？

重点

- 合理的代码风格和逗号放置可以使你的列表，字典，或集合更容易维护
- Python 的字符串字面连体功能可以带来好处，也可能引入难以捕捉的错误。

2.3 上下文管理器和with语句

有时候Python中的 `with` 语句被当做一种晦涩的难懂的特性。但是当你明白了其中的原理的时候，你会发现其中并没有什么魔法，并且实际上 `with` 语句这个特性可以帮你写出整洁和易读的代码。

那么`with`语句有什么用呢？通过抽象资源的功能，`with`语句有助于简化某些常见的资源管理模式并允许将它们分解和重用。

一个很好的观察这个功能的使用方法是查看Python标准库中的示例。内置的 `open()` 函数为我们提供了一个很好的用例：

```
with open('hello.txt', 'w') as f:
    f.write('hello, world!')
```

通常推荐使用`with`语句打开文件，因为它可以确保程序在执行`with`语句的上下文后自动关闭打开的文件描述符。最后，上面的代码示例可以转换为如下形式：

```
f = open('hello.txt', 'w')
try:
    f.write('hello, world')
```

```
finally:  
    f.close()
```

你可能会说这有点冗长。但是请注意 `try ... finally` 语句很重要。像下面这样写是不够的。

```
f = open('hello.txt', 'w')  
f.write('hello, world')  
f.close()
```

如果在 `f.write()` 调用期间发生异常，这种实现将不能保证文件一定被关闭，因此我们的程序可能导致文件描述符泄漏。这就是 `with` 语句如此有用的原因。它使获取和释放资源变得轻而易举。

另一个 `with` 语句被用于标准库的很好的例子是 `threading.Lock` 类：

```
some_lock = threading.Lock()  
# Harmful:  
some_lock.acquire()  
try:  
    # Do something...  
finally:  
    some_lock.release()  
  
# Better:  
with some_lock:  
    # Do something...
```

在这两种情况下，使用 `with` 语句可以抽象出大部分资源处理逻辑。不必每次都写一个 `try... finally` 语句，可以使用 `with` 语句帮我们做这些事情。

`with` 语句可以使处理系统资源的代码更具可读性。通过自动清理或释放不再需要的资源，还可以帮助你避免错误或资源泄漏。

在你自己的对象中支持 `with` 语句

`open()` 函数或 `threading.Lock` 类之所以可以使用 `with` 语句并不是因为任何特殊的地方或者魔法。你可以通过实现所谓的上下文管理器来为自己的类和函数提供相同的功能。

什么是上下文管理器？这是一个对象要支持 `with` 语句需要遵从的简单“协议”。如果你想让一个对象充当上下文管理器，你所需要做的就是在你的对象中添加 `_enter__` 和 `_exit__` 方法。Python 将在资源管理周期的适当时间调用这两个方法。

让我们看一下实际情况。一个简单实现的 `open()` 上下文管理器可能像下面这样：

```
class ManagedFile:  
    def __init__(self, name):  
        self.name = name  
  
    def __enter__(self):  
        self.file = open(self.name, 'w')  
        return self.file  
  
    def __exit__(self, exc_type, exc_val, exc_tb):  
        if self.file:  
            self.file.close()
```

就像原始的 `open()` 示例做的，现在我们的 `ManagedFile` 类遵循了上下文管理器协议，支持了 `with` 语句：

```
>>> with ManagedFile('hello.txt') as f:  
...     f.write('hello, world!')  
...     f.write('bye now')
```

当进入上下文时，Python 调用 `_enter__` 获取资源。当离开上下文时，Python 调用 `_exit__` 释放资源。

编写基于类的上下文管理器并不是唯一的支持 `with` 语句的方式。标准库中的 `contextlib` 模块提供了一些基本的上下文管理器协议之上的抽象。如果你的使用场景与 `contextlib` 所提供的相匹配，使用它会更简单一点。

例如，你可以使用 `contextlib.contextmanager` 装饰器为资源定义基于生成器的工厂函数，该函数将自动支持with语句。下面是我们使用这种技术重写的 `ManagedFile` 上下文管理器示例：

```
from contextlib import contextmanager
@contextmanager
def managed_file(name):
    try:
        f = open(name, 'w')
        yield f
    finally:
        f.close()

>>> with managed_file('hello.txt') as f:
...     f.write('hello, world!')
...     f.write('bye now')
```

在这种情况下，`managed_file()` 是一个生成器，它首先获取资源，之后，它会暂时中止自己的执行，在 `yield` 处产生资源，以便调用者可以使用它。当调用者离开 `with` 上下文，生成器继续执行，以执行剩余的清理步骤，将资源释放。

基于类的实现和基于生成器的实现本质上是等效的。你可能更喜欢其中的一个，具体取决于你认为哪种方法更具可读性。

基于 `@contextmanager` 的实现的缺点可能是它需要对 Python 的高级概念有一些了解，例如装饰器和生成器。如果你需要获取相关的知识，请随意阅读本书中的相关章节。

再强调一次，在这里做出正确的选择取决于你和你团队的习惯以及你们对易读性的评价标准。

使用上下文管理器编写漂亮的接口

上下文管理器非常灵活，如果你创造性地使用 `with` 语句，则可以为模块和类定义便捷的 API。例如，如果我们要管理的“资源”是某种报告生成器程序中的文本缩进级别，我们可以这样编写代码：

```
with Indenter() as indent:
    indent.print('hi!')
    with indent:
        indent.print('hello')
        with indent:
            indent.print('bonjour')
    indent.print('hey')
```

这几乎就像用于缩进文本的特定领域的语言（DSL）一样。另外，请注意代码如何多次进入和离开同一个上下文管理器以更改缩进级别。运行该代码段会得到以下输出并整齐地将格式化文本输出到控制台：

```
hi!
hello
    bonjour
hey
```

那么，你将如何实现上下文管理器来支持此功能？

顺便说一句，这可能是一个很好的练习机会，可以让你准确了解上下文管理器如何工作。因此，在你查看下面的实现之前，你最好将其作为一个练习，花一些时间尝试实现这个功能。

如果你准备看看我的实现，这里是一个使用基于类的上下文管理器实现此功能：

```
class Indenter:
    def __init__(self):
        self.level = 0

    def __enter__(self):
        self.level += 1
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.level -= 1
        def print(self, text):
            print(' ' * self.level + text)
```

那还不错，不是吗？我希望你已经感觉到在Python程序中使用with语句和上下文管理器会更加优雅。它们是一项出色的特性，可让你以更加符合Python风格的方式管理资源，并且让代码更易于维护。

如果你正在寻找另一个练习来加深理解，尝试实现一个使用time.time函数测量执行时间的上下文管理器。一定要尝试同时编写基于装饰器和基于类的变体以理解两者之间的区别。

重点

- with语句通过将 try/finally 语句的标准用法封装在上下文管理器中以简化异常处理。
- 最常见的场景是用于安全地获取和释放系统资源。进入with语句时自动获取资源离开with语句时自动释放资源。
- 有效使用上下文管理器可以帮助你避免资源泄漏并且使你的代码更易于阅读。

2.4 下划线，双下划线和其他

单下划线和双下划线在Python变量和方法具有特殊的意义。某些只是按照惯例向程序员提供一些提示，其中一些是Python解释器强制的。

如果你想知道，“Python变量和方法名称中的单下划线和双下划线有什么意思？”我会尽力给出答案。在本章中，我们将讨论以下五个下划线模式和命名约定，以及它们如何影响Python程序的行为：

- 单下划线开头的变量： `_var`
- 单下划线结尾的变量： `var_`
- 双下划线开头的变量： `__var`
- 双下滑先开头和结尾的变量： `__var__`
- 单下划线变量： `_`

1. 单下划线开头的变量： `_var`

单下划线开头的变量和方法仅具有约定的含义。这是对程序员的一种提示，这只是Python社区的一种约定，但这不会影响程序的行为。

单下划线前缀是用来提示这个变量或方法供内部使用。PEP 8（最常用的Python代码风格指南）中定义了这个约定。

但是，Python解释器没有强制执行这个约定。Python在“private”和“public”之间没有像Java一样明显的区别。在变量和方法名称前面添加单下划线更像是一个警告：“这并不是要成为此类的公共接口，最好不要直接调用它。”

看下面的例子：

```
'class Test:
def __init__(self):
    self.foo = 11
    self._bar = 23
```

如果你实例化该类并尝试访问该类在其`__init__`方法中定义的`foo`和`_bar`属性时会发生什么？

让我们试试：

```
>>> t = Test()
>>> t.foo
11
>>> t._bar
23
```

变量`_bar`中的前导下划线并没有阻止我们“进入”该类并获得它的值。

那是因为Python中的单下划线前缀只是一个约定俗成的惯例——至少在变量和方法名称中是如此。但是，从模块导入时下划线会影响导入的行为。假设你在一个名为`my_module`的模块中有如下代码：

```
# my_module.py:
def external_func():
    return 23
```

```
def _internal_func():
    return 42
```

现在，如果你使用通配符来从模块导入，Python不会导入带有下划线的名称（除非模块定义了`_all_`列表来覆盖此行为）：

```
>>> from my_module import *
>>> external_func()
23
>>> _internal_func()
NameError: "name '_internal_func' is not defined"
```

顺便说一句，应该避使用免通配符导入，因为它们导致名称空间混乱。为了清楚起见，最好坚持使用常规导入。与通配符导入不同，常规导入不受前导下划线命名的影响：

```
>>> import my_module
>>> my_module.external_func()
23
>>> my_module._internal_func()
42
```

我知道这可能有点令人困惑。如果你坚持PEP 8应该避免通配符导入的原则，那么你只需要记住：

单个下划线是Python的命名约定，表示名称仅供内部使用。它通常不由Python解释器强制校验，仅仅是对程序员的提示。

2. 单下划线结尾： `var_`

有时，某个变量最合适的名字已经被Python中的关键字占用。因此，`class`或`def`之类的名称不能在Python中用作变量名。在这种情况下，你可以通过附加单个下划线解决命名冲突：

```
>>> def make_object(name, class):
SyntaxError: "invalid syntax"
>>> def make_object(name, class_):
... pass
```

总之，使用单个下划线结尾来避免与Python关键字的命名冲突是一个约定，PEP 8中定义和解释了这个约定。

3. 双下划线开头： `__var`

到目前为止，我们讨论的命名规范都是一些约定俗成。使用以双下划线开头的Python类属性（变量和方法）时，事情就不一样了。

Python解释器会重写双下划线前缀的属性名称以避免子类中的命名冲突。

这也称为名称处理——当类在被扩展的时候解释器通过一种更不易出现名称冲突的方法更改变量名称。

我知道这听起来很抽象。所以我放了这一小段代码来做实验：

```
class Test:
    def __init__(self):
        self.foo = 11
        self._bar = 23
        self.__baz = 23
```

我们使用内建的`dir()`方法看看这个对象的属性：

```
>>> 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']
```

这为我们提供了包含对象属性的列表。让我们来看看这个清单找一下我们的原始变量名称 `foo`，`_bar` 和 `__baz`。我保证你会注意到一些有趣的变化。

首先，`self.foo` 变量在属性列表中还是 `foo`，没有什么变化。接下来，`self._bar` 也一样，它显示为 `_bar`。就像我之前说过的，下划线开头只是一个约定，是一种对程序员的提示。

但是，对于 `self.__baz` 来说，情况看起来有些不同。当你在该列表中搜索 `__baz`，你会看到没有变量有这个名字。

对于 `__baz` 来说，发生了什么？

如果仔细观察，会发现这个对象有一个名为 `_Test__baz` 的属性。这就是 Python 解释器的名称转换。这样做是为了防止变量被子类覆盖。

让我们创建另一个扩展 `Test` 类的类，并尝试覆盖其在初始化函数中添加的现有属性：

```
class ExtendedTest(Test):
    def __init__(self):
        super().__init__()
        self.foo = 'overridden'
        self._bar = 'overridden'
        self.__baz = 'overridden'
```

现在，你认为在这个 `ExtendedTest` 类的实例中，`foo`，`_bar` 和 `__baz` 的值会是什么？让我们来看看：

```
>>> t2 = ExtendedTest()
>>> t2.foo
'overridden'
>>> t2._bar
'overridden'
>>> t2.__baz
AttributeError:
"'ExtendedTest' object has no attribute '__baz'"
```

等等，为什么我们尝试获取 `t2.__baz` 的值时会抛出 `AttributeError`？名称被改变了！该对象甚至没有 `__baz` 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']
```

你看，`__baz` 变成了 `_ExtendedTest__baz` 以防止被意外修改。但是原来的 `_Test__baz` 仍然存在：

```
>>> t2._ExtendedTest__baz
'overridden'
>>> t2._Test__baz
42
```

双下划线名称改写对程序员是完全透明的。下面的示例将确认这一点：

```
class ManglingTest:
    def __init__(self):
        self.__mangled = 'hello'

    def get_mangled(self):
        return self.__mangled
>>> ManglingTest().get_mangled()
'hello'
>>> ManglingTest().__mangled
AttributeError:
"'ManglingTest' object has no attribute '__mangled'"
```

名称转换也适用于方法名称吗？当然！名称转换会影响类上下文中以两个下划线开头的所有名称：

```
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
```

这是名称转换行为的另一个示例，也许令人惊讶：

```
_MangledGlobal__mangled = 23
class MangledGlobal:
    def test(self):
        return __mangled

>>> MangledGlobal().test()
23
```

在此示例中，我将 `_MangledGlobal__mangled` 声明为全局变量。然后我在 `MangledGlobal` 类的上下文中访问了这个变量。由于名字改变，我得以在类的 `test()` 方法内使用 `__mangled` 引用 `_MangledGlobal__mangled` 全局变量。

因为 `__mangled` 以两个下划线开头，Python解释器会自动将 `__mangled` 转换到 `_MangledGlobal__mangled`。这表明名称转换不仅仅作用在类属性。它适用于在类上下文中以两个下划线字符开头的任何名称。

这要消化很多东西。

老实说，我没有不假思索就写下这些例子。我花了一些时间研究才做到这一点。我已经使用Python多年了，但是像这样的规则和特殊情况我也做不到信手拈来。

有时，对于程序员而言，最重要的技能是“模式识别”，知道在哪里查找相关的答案。如果你此时有一点感觉不堪重负，别担心。花时间把玩一下本章中的一些示例。

让这些概念足够深入，你才能够认识到名称转换的理念以及我展示的一些其他行为。如果有一天你猛然遇到它们，你会知道在文档中寻找什么。

什么是dunders

如果你听说过一些经验丰富的Python使用者谈论Python或观看过一些会议演讲，你可能已经听说过dunder一词。如果你想知道这是什么，那么，答案在这里：

双下划线在Python中通常称为“dunder”。原因是双下划线出现的频率很高，为了更容易发音，Python开发者通常将“dunder”作为“double underscore”的缩写。

例如，`__baz` 可以发音为“dunder baz”。同样地，`__init__` 可以发音为“dunder init”，即使有人可能认为它应该是“dunder init dunder”。但这只是另一个命名约定。就像是Python开发人员的秘密握手。

4. 双下划线开头和结尾：`__var__`

也许令人惊讶的是，如果名称以双下划线开头并以双下划线结尾，则不会进行名称转换。Python解释器没有对双下划线开头和双下划线结尾的变量进行任何修改：

```
class PrefixPostfixTest:
    def __init__(self):
        self.__bam__ = 42
>>> PrefixPostfixTest().__bam__
42
```

但是，双下划线开头和双下划线结尾的名称被语言保留以供特殊使用。像 `__init__` 方法、`__call__` 方法都符合这个规则。

这些双下划线开头和结尾的方法通常被称为魔术方法，但是在Python社区中，包括我自己在内的许多人都不喜欢这个词。这个词意味着不鼓励使用双下划线开头和结尾的方法，但是事实完全不是这样。它们是Python的一个核心特性，应该根据需要加以使用。他们并没有什么“魔法”。

但是，就命名约定而言，在自己的代码中最好不要使用以双下划线开头和结尾的名称，以避免与将来与Python语言的更改产生冲突。

5.单下划线： _

按照惯例，有时使用单个独立的下划线表示变量是临时的或无关紧要的。例如，在以下循环中，我们不需要访问正在运行的索引，我们可以使用“_”表示它只是一个临时值：

```
>>> for _ in range(32):
... print('Hello, World.')
```

你还可以在解包语句中使用单个下划线来忽略无关紧要的特定值。同样，这也仅仅是一个惯例，并不会触发Python解释器的任何特殊行为。单下划线是一个有效变量名，只是有时候被这样用了而已。

在下面的代码示例中，我将一个元组解包为单独的变量，我只对颜色和里程字段感兴趣。但是，为了使解包表达式能够正常工作，我需要将元组中包含的所有值分配给变量。这就是占位符变量“_”的作用：

```
>>> car = ('red', 'auto', 12, 3812.4)
>>> color, _, _, mileage = car
>>> color
'red'
>>> mileage
3812.4
>>> _
12
```

除了用作临时变量外，“_”在大多数情况下是表示解释器执行的最后一个表达式结果的特殊变量。如果你正在解释器的会话中工作并且想访问先前计算的结果，这会很方便：

```
>>> 20 + 3
23
>>> _
23
>>> print(_)
23
```

如果你正在动态地构建对象并且想要在不给他们分配名称的情况下与其进行交互，这也很方便：

```
>>> list()
[]
>>> _.append(1)
>>> _.append(2)
>>> _.append(3)
>>> _
[1, 2, 3]
```

重点

- 单个前导下划线“`_var`”：命名惯例，表示名称仅供内部使用。通常不由Python解释器强制执行（通配符导入除外）只是对程序员的提示。
- 单尾划线“`var_`”：按惯例用于避免与Python关键字冲突。
- 双引号下划线“`__var`”：在类上下文中使用时会触发名称转换，由Python解释器强制执行。
- 前后双下划线“`__var__`”：表示由Python语言定义的特殊方法。避免自己的属性使用这种命名方案。
- 单个下划线“`_`”：有时用作临时或无关紧要的变量的名称。此外，它还表示在Python REPL会话中发送最后一个表达式的结果。

2.5 关于字符串格式化的真相

还记得在Python之禅有一条“做某件事应该有一种明显的方式？”当你发现在Python中有四种主要的方法可以进行字符串格式化时，你可能会抓狂。

在本章中，我将演示这四种字符串格式化方法的使用方法以及它们各自的优缺点。我还将给出我如何选择最佳字符串格式化方法的“经验法则”。

因为我们有很多内容要讲，所以我们直接开始。为了有一个简单的实验示例，假设我们有以下变量可以使用：

```
>>> errno = 50159747054
>>> name = 'Bob'
```

根据这些变量，我们想输出带有以下错误消息字符串：

```
'Hey Bob, there is a 0xbadc0ffee error!'
```

现在，该错误可能真的浪费了开发人员星期一的早上！但是我们今天是在这里讨论字符串格式。因此，让我们开始工作。

#1 – 老式的字符串格式化

Python中的字符串具有唯一的内置操作标识符，可以使用%号访问。这是实现简单的位置格式化的捷径。如果你曾经使用过C语言中的printf函数，你会立即意识到它是如何工作的。这是一个简单的示例：

```
>>> 'Hello, %s' % name
'Hello, Bob'
```

我在这里使用%s来告诉Python在哪里将其替换成以字符串表示的值。这就是所谓的“老式”字符串格式化。

在“老式”字符串格式化中，还有其他格式说明符可用于控制输出的字符串。例如，可以将数字转换为十六进制表示法或添加空格以生成格式正确的表格和报告。

在这里，我使用%x格式说明符将int值转换为十六进制数字并用字符串表示出来：

```
>>> '%x' % errno
'badc0ffee'
```

如果你想在单个字符串中进行多个替换，“老式”字符串格式语法会稍有变化。因为%操作符只需要一个参数，你需要将右侧的参数包裹在一个元组中，像下面这样：

```
>>> 'Hey %s, there is a 0x%x error!' % (name, errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

如果你将一个字典传递给%操作符，你还可以在字符串格式化中使用变量名指定变量：

```
>>> 'Hey %(name)s, there is a 0x%(errno)x error!' % {
... "name": name, "errno": errno }
'Hey Bob, there is a 0xbadc0fee error!'
```

这会使字符串格式化更易于维护和修改。你不用担心传递的值顺序与字符串格式化中引用顺序的匹配问题。当然，缺点是需要打更多的字。

我敢肯定你一直想知道为什么这种printf风格的格式化被打上“老式”的标签。老式的字符串格式化在技术上已被我们将要谈到的“新式”格式化取代。但是，虽然不再被强调，“老式”格式化并未被弃用。最新版本的Python仍支持该功能。

#2 新式字符串格式化

Python 3引入了一种新的字符串格式化方式，后来也支持了Python 2.7。这种“新式”字符串格式化弃用了%操作符，让字符串格式化更规整。现在可以通过在字符串对象上调用format()函数来进行格式化。

你可以使用format()函数进行简单的基于位置的格式化，这和使用“老式”格式化很像：

```
>>> 'Hello, {}'.format(name)
'Hello, Bob'
```

你在按名称进行变量替换时，可以按照任何你喜欢的顺序进行。这是一个非常强大的功能，因为它允许重新排列显示顺序而不用更改传递给format函数的参数：

```
>>> 'Hey {} , there is a 0x{} error!'.format(
...     name=name, errno=errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

将int变量格式化为十六进制字符串的语法也有变化。现在通过在变量名后添加“: x”后缀，我们添加一个了格式化规范。

总体而言，新式字符串格式化在没有复杂化简单用例的前提下，让字符格式化语法变得更加强大。Python文档中关于[字符串格式化](#)的部分值得一读。

在Python 3中，这种“新式”字符串格式化更受青睐。但是，从Python 3.6开始，还有一种更好的字符串格式化方法。在下一节中，我将告诉你所有的相关信息。

#3 文字字符串内插(Python 3.6+)

Python 3.6添加了另一种格式化字符串的方式，称为格式化的文字字符串。这种格式化字符串的新方法使你可以在字符串常量中嵌入Python表达式。这是一个简单的示例，可以让你对该功能有所了解：

```
>>> f'Hello, {name}!'
'Hello, Bob!'
```

这种新的格式语法功能强大。由于你可以嵌入任意Python表达式，因此甚至可以进行运算，比如：

```
>>> a = 5
>>> b = 10
>>> f'Five plus ten is {a + b} and not {2 * (a + b)}.'
'Five plus ten is 15 and not 30.'
```

格式化的字符串文字依赖Python解析器的一个特性，这个特性可以将f字符串转换为一系列字符串常量和表达式。然后，他们会被拼接起来构建最终的字符串。

假设我们有一个包含f字符串的greet()函数：

```
>>> def greet(name, question):
...     return f"Hello, {name}! How's it {question}?"
...
>>> greet('Bob', 'going')
"Hello, Bob! How's it going?"
```

当我们反汇编该函数并对其进行检查时，我们可以看到该函数中的f字符串已经被转换为类似于下面的东西：

```
>>> def greet(name, question):
...     return ("Hello, " + name + "! How's it " +
            question + "?")
```

实际的实现要比上面的实现快一点，因为它使用了[BUILD_STRING](#)进行优化。但从功能上讲，它们是相同的：

```
>>> import dis
>>> dis.dis(greet)
2      0 LOAD_CONST 1 ('Hello, ')
2      2 LOAD_FAST 0 (name)
4      4 FORMAT_VALUE 0
6      6 LOAD_CONST 2 ('! How's it ')
8      8 LOAD_FAST 1 (question)
10    10 FORMAT_VALUE 0
12    12 LOAD_CONST 3 ('?')
```

```
14 BUILD_STRING 5
16 RETURN_VALUE
```

文字字符串也支持现有的str.format()语法。这样你就可以解决我们在前两节中讨论的格式化问题：

```
>>> f"Hey {name}, there's a {errno:#x} error!"
"Hey Bob, there's a 0xbadc0ffee error!"
```

Python的新格式化字符串文字类似于JavaScript在ES2015中添加的文字模板。我认为它们是Python很好的一个补充，并且我已经在日常的Python 3工作中开始使用它们。你可以在Python官方文档中了解有关格式化字符串的更多信息。

#4-字符串模板

在Python中用于字符串格式化的另一种技术是字符串Template。这是一种更简单，功能没有那么强大的机制，但在某些情况下，这可能正是你所寻找的特性。

让我们看一个简单的用于问候示例：

```
>>> from string import Template
>>> t = Template('Hey, $name!')
>>> t.substitute(name=name)
'Hey, Bob!'
```

在这里我们需要从Python的内置模块导入字符串模版。模板字符串不是语言的核心功能，但由标准库中的一个模块提供。

另一个区别是模板字符串不允许格式标识符。因此，为了使错误显示字符串正常输出，我们需要自己将整形错误号转换为十六进制字符串：

```
>>> templ_string = 'Hey $name, there is a $error error!'
>>> Template(templ_string).substitute(
...     name=name, error=hex(errno))
'Hey Bob, there is a 0xbadc0ffee error!'
```

效果很好，你可能想知道应该在什么时候使用模板字符串。我认为，模板字符串最好的使用场景是处理用户生成的格式字符串。由于降低了复杂度，模板字符串是一个更安全的选择

其他更复杂的字符串格式化技术可能会在你的程序中引入安全漏洞。例如，格式字符串可以访问程序中的任意变量。

这意味着，如果恶意用户可以提供格式字符串，他们就有可能泄露密钥和其他敏感信息！这是关于如何使用这种攻击的一个简单例子：

```
>>> SECRET = 'this-is-a-secret'
>>> class Error:
...     def __init__(self):
...         pass
>>> err = Error()
>>> user_input = '{error.__init__.globals__[SECRET]}'
# Uh-oh...
>>> user_input.format(error=err)
'this-is-a-secret'
```

了解潜在的攻击者是怎么从格式字符串访问__globals__字典来获取我们的SECRET字符串了吗？吓死了，呵呵！模板字符串阻断了这种攻击方式，所以如果你要将用户输入进行字符格式化，字符串模板是一个更安全的选择：

```
>>> user_input = '${error.__init__.globals__[SECRET]}'
>>> Template(user_input).substitute(error=err)
ValueError:
"Invalid placeholder in string: line 1, col 1"
```

我应该使用哪种字符串格式化方法？

我完全明白，在Python中有这么多格式化字符串的方式可供选择会让人感到非常混乱。这时最好列出一些流程图...

但是我不会那样做。相反地，我会尝试将其归结为我在编写Python代码时遵循的一些简单经验法则。

你可以在不清楚选择那种字符串格式化方法的时候，根据具体情况使用这些经验。

笔者的Python字符串格式化经验

如果你的格式字符串是用户提供的，请使用字符串模板以避免出现安全问题。如果你使用的是Python 3.6及更高版本，请使用字符串内插，如果不是，使用“新式”字符串格式化。

重点

- Python有不止一种格式化字符串的方式。
- 每种方法都有其各自的优缺点。应该使用哪种方法应该根据你的使用场景确定。
- 如果你在确定使用哪种字符串格式化方法时遇到麻烦，请尝试我的“字符串格式化经验法则”

2.6 Python之禅

我知道，就Python书籍而言，有下面的内容是很常见的。但Tim Peters的Python之禅真的无法绕过。多年以来通过时常重新审视Python之禅，让我受益匪浅，Tim的话使我成为更好的程序员，希望你也一样。

此外，Python Zen确实很重要，因为它包含在语言的复活节彩蛋中。只需在Python解释器输入并运行以下命令：

```
>>> import this
```

Python之禅

优美胜于丑陋

明了胜于晦涩

简洁胜于复杂

复杂胜于凌乱

扁平胜于嵌套

间隔胜于紧凑

可读性很重要

即便假借特例的实用性之名，也不可违背这些规则

不要包容所有错误，除非你确定需要这样做

当存在多种可能，不要尝试去猜测

而是尽量找一种，最好是唯一一种明显的解决方案

虽然这并不容易，因为你不是 Python 之父

做也许好过不做，但不假思索就动手还不如不做

如果你无法向人描述你的方案，那肯定不是一个好方案

如果你很容易向人描述你的方案，那可能是一个好方案

命名空间是一种绝妙的理念，我们应当多加利用

source/python_tricks/3_函数.md

第三章 函数

3.1 Python的函数是一类对象

Python的函数是一类对象。你可以将它们分配给变量，将它们存储在数据结构中，将它们作为参数传递给其他函数，甚至将它们作为其他函数的返回值。

理解这些概念将使你在深入理解Python中lambda和装饰器这类高级功能时更轻松。它还能使你接触函数式编程的技术。

在接下来的几页中，我将指导你完成许多示例以帮助你加深理解。这些例子一环扣一环，因此你可能需要按顺序阅读它们，甚至需要尝试其中的一些例子。

消化我们将在这里讨论的概念可能会花费比预期更长的时间。不用担心，那完全正常。我经历过，你可能会觉得就像自己把头撞在了墙上，然后突然之间事情会在你准备好后，回到适当的位置。

在本章中，我将使用此yell函数进行演示。这是一个简单的示例：

```
def yell(text):
    return text.upper() + '!'
>>> yell('hello')
'HELLO!'
```

函数是对象

Python程序中的所有数据均由对象或表示对象之间的关系代表。诸如字符串，列表，模块和函数都是对象。在Python中，函数没有什么特别之处，它们也只是对象。

由于yell函数是Python中的对象，因此你可以像其他任何对象一样，将其赋值给另一个变量：

```
>>> bark = yell
```

该行不会调用该函数。它获取yell引用的函数对象，并创建指向它的第二个名称bark。你现在也可以通过调用bark执行相同的函数对象：

```
>>> bark('woof')
'WOOF!'
```

函数对象及其名称是两个单独的概念。你可以删除函数的原始名称yell，另一个名称bark仍指向原来的函数，你仍然可以通过bark调用函数：

```
>>> del yell
>>> yell('hello?')
NameError: "name 'yell' is not defined"
>>> bark('hey')
'HEY!'
```

顺便说一句，为了调试方便，Python在创建函数时，为每个函数添加了一个字符串标识符。你可以通过`_name_`属性访问此标识符：

```
>>> bark._name_
'yell'
```

现在，虽然函数的`_name_`属性仍然是yell，但这并不影响从代码访问函数对象。名称标识符仅仅用于辅助调试。指向函数的变量和函数本身实际上是两个独立的点。

函数可以被保存在数据结构中

由于函数是一等公民，因此你可以像处理其他对象一样，将其存储在数据结构中。例如，你可以将函数添加到列表中：

```
>>> funcs = [bark, str.lower, str.capitalize]
>>> funcs
[<function yell at 0x10ff96510>,
<method 'lower' of 'str' objects>,
<method 'capitalize' of 'str' objects>]
```

访问存储在列表中的函数对象和访问任何其他类型的对象没有区别：

```
>>> for f in funcs:
...     print(f, f('hey there'))
<function yell at 0x10ff96510> 'HEY THERE!'
<method 'lower' of 'str' objects> 'hey there'
<method 'capitalize' of 'str' objects> 'Hey there'
```

你甚至可以调用存储在列表中的函数对象，而无需将其先赋值给一个变量。你可以进行查找，然后立即在单个表达式中调用返回的函数对象：

```
>>> funcs[0]('heyho')
'HEYHO!'
```

函数可以当做参数传给其它函数

因为函数是对象，所以你可以将它们作为参数传递给其他函数。这个greet函数用于格式化问候字符串，在期间会调用传递给它的函数对象，然后打印它：

```
def greet(func):
    greeting = func('Hi, I am a Python program')
    print(greeting)
```

你可以通过传递不同的函数来影响产生的结果。如果你传入bark函数，结果会像下面这样：

```
>>> greet(bark)
'HI, I AM A PYTHON PROGRAM!'
```

当然，你也可以定义一个新函数来生成不同的问候语。比如，如果你不希望你的Python程序听起来听起来像擎天柱，下面的whisper函数可能会更好：

```
def whisper(text):
    return text.lower() + '...'
>>> greet(whisper)
'hi, i am a python program...'
```

将函数对象作为参数传递给其他函数是很强大的功能。它可以在程序中抽象并绕过其中的行为。在这里示例中，greet函数保持不变，但你可以通过传递不同的问候行为来影响其输出。

可以接受其他函数作为参数的函数也称为高阶函数。它们是函数式编程的必要条件。

Python中高阶函数的经典示例是内置的map函数。其参数是一个函数对象和一个可迭代对象，然后随着迭代进行，它在每个可迭代的元素上调用函数、产生结果。

你也可以通过将bark函数进行mapping,一次性格式化一系列问候语：

```
>>> list(map(bark, ['hello', 'hey', 'hi']))
['HELLO!', 'HEY!', 'HI!']
```

你看，map遍历了整个列表，并将bark函数应用于每个元素。现在我们有了一个新列表。

函数可以嵌套

也许会令人惊讶，Python允许在函数内部定义函数。这些通常称为嵌套函数或内部函数。这是一个例子：

```
def speak(text):
    def whisper(t):
        return t.lower() + '...'
    return whisper(text)
>>> speak('Hello, World')
'hello, world...'
```

这是怎么回事？每次调用speak函数时，它都会定义一个新的内部函数whisper，然后立即调用它。我在这里理解得有点慢，但是总的来说，这仍然是相对简单的东西。

speak函数外部不存在whisper：

```
>>> whisper('Yo')
NameError:
"name 'whisper' is not defined"
>>> speak.whisper
AttributeError:
"'function' object has no attribute 'whisper'"
```

但是，如果你真的想从speak函数外访问该嵌套的whisper函数，该怎么办？好吧，函数就是对象，你可以返回内部函数给父函数的调用者。

例如，这是一个定义了两个内部函数的函数。根据传递给外层函数的参数，它选择并返回其中一个内部函数给调用者：

```
def get_speak_func(volume):
    def whisper(text):
        return text.lower() + '...'
    def yell(text):
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper
```

请注意，get_speak_func实际上并未调用其内部的任何函数——它仅根据volume参数选择了适当的内部函数并将其返回：

```
>>> get_speak_func(0.3)
<function get_speak_func.<locals>.whisper at 0x10ae18>
>>> get_speak_func(0.7)
<function get_speak_func.<locals>.yell at 0x1008c8>
```

当然，你可以直接调用返回的函数，或者首先将其分配给变量然后再进行调用：

```
>>> speak_func = get_speak_func(0.7)
>>> speak_func('Hello')
'HELLO!'
```

先静下心来想一想... 这意味着函数不仅可以通过参数接受行为，还可以返回行为。多么酷啊！

你知道吗，到这里后，事情开始有些混乱了。我要在继续写作之前先去喝杯咖啡休息一下（我建议你也来一杯）

函数可以捕获局部状态

你刚刚看到了函数如何包含内部函数，并且函数还可以从父函数返回这些内部函数。

现在最好系上安全带，因为后面会有点疯狂。我们将进入更深层次的函数编程领域。（你有喝咖啡休息，对吧？）

函数不仅可以返回其他函数，而且这些内部函数也可以捕获并携带某些父函数的状态。好吧，那是什么意思呢？

我重写一下之前的get_speak_func示例来说明这一点。新版本带有一个“volume”和一个“text”参数，以使返回的函数可立即调用：

```

def get_speak_func(text, volume):
    def whisper():
        return text.lower() + '...'
    def yell():
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper

>>> get_speak_func('Hello, World', 0.7)()
'HELLO, WORLD!'

```

现在，仔细看看内部函数whisper和yell。注意他们为何不再需要text参数？但是他们仍然可以访问父函数中定义的text参数。实际上，他们似乎捕获并“记住”了这个参数的值。

执行此操作的函数称为词法闭包（简称闭包）。即使程序执行已经不在该范围内，闭包也会记住其封闭词法中的值。

实际上，这意味着函数不仅可以返回行为它们也可以预先配置这些行为。这里有另一个简单例子来说明这个想法：

```

def make_adder(n):
    def add(x):
        return x + n
    return add
>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)
>>> plus_3(4)
7
>>> plus_5(4)
9

```

在此示例中，make_adder用作创建和配置“adder”函数的工厂。注意观察adder函数仍然可以访问make_adder函数的参数n（封闭范围）。

对象可以表现得像函数

尽管所有函数都是Python中的对象，但反之并不成立。对象不是函数，但是可以将它们设为可调用的，这允许你在许多情况下将它们视为函数。

如果一个对象是可调用的，则意味着你可以使用圆括号函数调用语法，甚至传递函数调用参数。所有这些都由__call__方法实现。这是定义可调用对象的示例：

```

class Adder:
    def __init__(self, n):
        self.n = n
    def __call__(self, x):
        return self.n + x
>>> plus_3 = Adder(3)
>>> plus_3(4)
7

```

实际上，“调用”对象会尝试执行对象的__call__方法。

当然，并非所有对象都是可调用的。这就是为什么会有内置的用于检查对象是否看起来可调用的callable函数：

```

>>> callable(plus_3)
True
>>> callable(yell)
True
>>> callable('hello')
False

```

重点

Python中包括函数在内的所有内容都是一个对象。你可以将它们赋值给变量，将它们存储在数据结构中，将它们传递或返回给其他函数（或从其他函数返回）。

- 一类的函数使你可以抽象和传递行为。
- 函数可以嵌套，并且可以捕获并携带一些父函数的状态。这个功能被称为闭包。
- 可以使对象像函数一样可调用。在许多情况下，这使你可以像对待函数一样对待他们。

3.2 lambda是单表达式函数

Python中的lambda关键字提供了声明小型匿名函数的快捷方式。Lambda函数的行为类似于用def关键字声明的常规函数。需要函数对象时，可以使用lambda函数。

例如，这是你定义的用于执行加法的简单lambda函数：

```
>>> add = lambda x, y: x + y
>>> add(5, 3)
8
```

你可以用def关键字声明相同的add函数，但会更加冗长：

```
>>> def add(x, y):
    return x + y
>>> add(5, 3)
8
```

现在你可能想知道，lambda有什么特别的地方？如果它们只是用def声明函数的简明版本，有什么大不了的？

请看以下示例，在看的时候将“函数表达式”这个词记在脑海中：

```
>>> (lambda x, y: x + y)(5, 3)
8
```

好的，这是怎么回事？我只是使用lambda定义了一个add函数，然后使用参数5和3立即调用这个函数。

从概念上讲，lambda表达式lambda x, y: x + y与使用def声明的函数没有区别，它只是写在一行中。主要区别在于在使用之前我不必分配一个名字给匿名函数对象。我只是在lambda表达式中描述我的计算逻辑，然后像调用常规函数一样立即调用lambda表达式。

在继续之前，为了真正理解其含义，你可能需要把玩一下之前的代码。我仍然记得我花了多少时间来理清这些东西。所以不用担心在一个解释器的会话窗口上多花了几分钟，这是值得的。

lambda函数和常规函数之间还有另一个语法差异。Lambda函数仅限于一个表达式。这意味着lambda函数不能使用语句或注释——甚至没有return语句。

那么如何从lambda返回值？执行lambda函数时会计算其表达式的值，然后自动返回计算结果，因此总会有一个隐式的返回值。这就是为什么有些人将lambda称为单个表达式函数。

你可以使用的lambda表达式

什么时候应该在代码中使用lambda函数？从技术上讲，在任何你期望提供一个函数对象的时候，都可以使用lambda表达式。而且因为lambda是匿名的，你甚至都不需要首先为其分配一个名称。

这提供了一个方便简洁的快捷方式来定义Python中的函数。我最常使用lambda的场景是编写简洁的key函数，通过提供key函数对可迭代对象进行排序：

```
>>> tuples = [(1, 'd'), (2, 'b'), (4, 'a'), (3, 'c')]
>>> sorted(tuples, key=lambda x: x[1])
[(4, 'a'), (2, 'b'), (3, 'c'), (1, 'd')]
```

在上面的示例中，我们按第二个值对元组列表进行排序。在这种情况下，lambda函数提供了一种快速的方法修改排序顺序。这是你可以玩的另一个排序示例：

```
>>> sorted(range(-5, 6), key=lambda x: x * x)
[0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5]
```

我向你展示的两个示例使用Python内置的operator.itemgetter()和abs()函数都有更简洁的实现。但我希望你能看到使用lambda会给你带来更大的灵活性。要按一个任意的key对序列进行排序？没问题，现在你知道该怎么做了。

这是关于lambda的另一件有趣的事情：就像常规的嵌套函数一样，lambda还可作为词法闭包。

什么是词法闭包？它只是一个函数的奇特名称，表示即使程序已经不再在该范围内，但是还是会记住来自封闭词法范围的值。这是一个说明这个想法的例子：

```
>>> def make_adder(n):
    return lambda x: x + n

>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)
>>> plus_3(4)
7
>>> plus_5(4)
9
```

在上面的示例中，即使n是在make_adder函数中定义的（闭包范围），x + n lambda表达式仍可以访问n的值。

有时，使用Lambda函数而不是用def关键字声明的嵌套函数可以更清楚地表达程序员的意图。但是，老实说，这并不常见——至少这不是我喜欢的代码风格。所以我们再谈一点关于这个话题的信息。

或许你不应该...

一方面，我希望本章除能使你对探索Python的lambda函数感兴趣。另一方面，我觉得是时候提出另一个警告了：应谨慎使用Lambda函数。

我知道我已经使用lambda编写了相当一部分“很酷”的代码，但实际上这对我和我的同事来说是个负担。如果你倾向于使用lambda，花几秒钟（或几分钟）思考一下，这是否是实现这一目标的最简洁、维护性最好的方法。

例如，做下面这样的事情来节省两行代码是愚蠢的。当然，从技术上讲，它是可行的，并且有点炫技。但这也让下一个必须在一个很紧凑的期限内修正错误的同事感到困惑：

```
# Harmful:
>>> class Car:
...     rev = lambda self: print('Wroom!')
...     crash = lambda self: print('Boom!')
>>> my_car = Car()
>>> my_car.crash()
'Boom!'
```

我对使用lambda的复杂map()或filter()函数有类似的感觉。通常，使用列表表达式或生成器表达式会更简洁：

```
# Harmful:
>>> list(filter(lambda x: x % 2 == 0, range(16)))
[0, 2, 4, 6, 8, 10, 12, 14]
# Better:
>>> [x for x in range(16) if x % 2 == 0]
[0, 2, 4, 6, 8, 10, 12, 14]
```

如果你发现自己在使用lambda表达式做很复杂的事情，请考虑定义一个具有合适名称的独立函数来实现。

从长远来看，节省几行代码无关紧要，但是你的同事（以及未来的你）会喜欢清晰易读的代码而不是一些奇技淫巧。

重点

- Lambda函数是单表达式函数，不必绑定到名称（匿名）。
- Lambda函数不能使用常规的Python语句，并且总是包含隐式return语句。
- 始终问自己：使用常规函数或列表解析是否会让代码更清晰？

3.3 装饰器的威力

Python的装饰器可让你扩展和修改其核心可调用对象（函数，方法和类）的行为，而无需永久修改可调用对象本身。

任何可以附加到现有类或函数的足够通用的功能都非常适用于使用装饰器实现。这包括以下内容：

- 日志
- 实施访问控制和身份验证
- 说明和计时
- 限速
- 缓存等

为什么要掌握Python中的装饰器呢？毕竟，我刚才提到的内容听起来很抽象，可能很难理解装饰器如何让Python程序员在日常工作中受益。让我通过给你一个真实的例子来设法使这个问题更加清楚：

想象一下，你的报表生成程序有30个涉及具体业务逻辑的函数。一个多雨的星期一早上，你的老板走到你办公桌前说：“星期一快乐！记得那些TPS报告吗？你需要你将输入/输出日志记录添加到报告生成工具。XYZ公司需要用它进行审计。哦，我告诉他们我们可以在星期三之前完成。

根据你是否深刻理解Python装饰器，这个需求可能会让你压力山大也可能让你平静如常。

如果没有装饰器，你可能会在接下来的三天时间里手动修改这30个功能中的每一个，并添加一堆杂乱无章的日志调用。好玩吗？

但是，如果你知道装饰器，你就可以冷静地微笑着对老板说：“别担心，吉姆，我会在今天下午2点之前完成。”

此后，你将编写一个通用的@audit_log装饰器（仅约10行），然后将其快速粘贴到每个函数定义代码的前面。然后提交代码并拿起另一杯咖啡…

我在这里有点戏剧化。装饰器就是可以这么强大。我要说的是，对任何认真的Python程序员来说，了解装饰器都是一个里程碑。他们需要扎实地掌握语言中的几个高级概念。

我相信，在Python中，理解装饰器的收益是巨大的

当然，开始的时候装饰器确实有点难以理解，但是它们是非常有用的功能，你在第三方框架和Python标准库中会经常遇到装饰器。解释装饰器也是任何Python教程成败的关键。我会尽力在这里向你介绍他们。

在开始学习之前，现在是重新复习一下Python一类函数属性的绝佳时机。本书中有一章介绍了这些内容，我鼓励你花几分钟时间进行复习。了解装饰器需要知道的最重要的“一类函数”的要点是：

- 函数是对象，可以将它们分配给变量和传递给其他函数并从其他函数返回
- 可以在其他函数中定义函数，并且子函数可以捕获父函数的局部状态（词法闭包）

好了，你准备好了吗？让我们开始吧。

Python装饰器基础

现在，装饰器到底是什么？他们“装饰”或“包裹”另一个函数，并让你在被装饰函数运行之前和之后执行代码。

装饰器允许你定义可重复使用的代码块，这些代码块可以更改或扩展其他函数的行为。而且，装饰器让你在不修改被装饰函数本身的前提下执行此操作。该函数的行为仅在装饰后才会改变。

一个简单的装饰器的实现会是什么样子？简单来说，装饰器是一个可调用对象，它以可调用对象为输入，返回另一个可调用对象。

以下函数就是这样，这可能是你可以写出的最简单的装饰器：

```
def null_decorator(func):
    return func
```

你看，null_decorator是可调用的（它是一个函数），它接受另一个可调用对象作为其输入，并原样返回相同的可调用对象。

让我们用它来装饰（或包装）另一个函数：

```
def greet():
    return 'Hello!'
```

```

greet = null_decorator(greet)

>>> greet()
'Hello!'

```

在此示例中，我定义了一个greet函数，然后通过运行null_decorator函数来装饰它。我知道这看起来不太有用。我的意思是，我们是故意将null_decorator装饰器设计为无用的，对吗？但是稍后，这个例子将充分说明Python装饰器语法的工作原理。

你可以使用Python的@语法更方便地装饰函数，而不必在greet上显式调用null_decorator函数然后重新赋值greet变量。

```

@null_decorator
def greet():
    return 'Hello!'

>>> greet()
'Hello!'

```

在函数定义的前面放置一个@null_decorator行和先定义函数然后调用null_decorator函数没有区别。@语法只是一个语法糖，是这种常用模式的一个简写。

请注意，使用@语法会在函数定义时立即装饰函数。这使得在没有使用黑魔法的情况下很难访问未经装饰的原始函数。因此，为了保持调用未修饰函数的功能，你可能会选择手动执行某些函数装饰器。

装饰器可以修改行为

现在，你对装饰器语法有了更多的了解，让我们编写另一个装饰器，该装饰器实际上会执行某些操作并修改被装饰函数的行为。

这是一个稍微复杂一点的装饰器，它可以将被装饰函数的输出转换为大写：

```

def uppercase(func):
    def wrapper():
        original_result = func()
        modified_result = original_result.upper()
        return modified_result
    return wrapper

```

这个uppercase装饰器不像null_decorator装饰器那样简单地返回输入函数，而是动态定义了一个新函数（闭包）并使用它来包装输入函数，以修改其调用时的行为。

wrapper闭包可以访问未修饰的输入函数，并且在调用输入函数之前和之后可以自由执行其他代码。（从技术上讲，它甚至不需要调用输入函数）

请注意，到目前为止，被装饰函数是从未执行过的。实际上，这时调用输入函数不会有任何意义——你希望装饰器在最终被调用时能够修改其输入函数的行为。

你可能想花一两分钟消化一下这些知识点。我知道这些东西看起来有多复杂，但我们终会将它们整理在一起的，我承诺。

是时候看看uppercase装饰器的作用了。如果用它装饰开始的greet函数会发生什么？

```

@uppercase
def greet():
    return 'Hello!'

>>> greet()
'HELLO!'

```

我希望这是你期望的结果，让我们仔细看看刚刚发生了什么。与null_decorator不同，我们的uppercase装饰器在装饰函数时返回了一个不同的函数对象：

```

>>> greet
<function greet at 0x10e9f0950>

>>> null_decorator(greet)
<function greet at 0x10e9f0950>

```

```
>>> uppercase(greet)
<function uppercase.<locals>.wrapper at 0x76da02f28>
```

如你先前所见，它需要这样做才能修改被装饰函数在最终被调用时的行为。`uppercase` 装饰器本身就是一个函数。影响它装饰的输入函数“未来行为”的唯一方法是使用一个闭包替换输入函数。

这就是为什么 `uppercase` 装饰器需定义并返回另一个函数（闭包）的原因，这个函数在后面可以被调用，它会运行原始的输入函数，并修改它的结果。

装饰器可以修改可调用对象的行为，因此你不必直接修改原始对象。装饰时，原始可调用对象没有被永久修改——仅其行为发生了变化。

这样一来，你就可以在现有的函数和类上添加可复用的代码块，比如日志记录和其他工具。装饰器具有如此强大的功能，所以在标准库和第三方程序中装饰器都被大量使用。

快速中场休息

顺便说一句，如果你感觉需要喝杯咖啡休息一下或走一走，这是完全正常的。在我看来，闭包和装饰器是Python中最难理解的一些概念。

请花些时间，不要想着立即解决这些问题。在解释器中一个一个把玩一下示例代码有助于加深理解。

我知道你可以做到的。

在函数上使用多个装饰器

也许不足为奇，你可以将多个装饰器应用于一个函数。装饰器的效果会累积，作为可重复使用的代码块，这是装饰器如此有用的原因。

这有一个例子。以下两个装饰器将被装饰函数的字符串包在HTML标签中。通过查看标签如何嵌套，你可以看到使用多个装饰器时装饰器的应用顺序：

```
def strong(func):
    def wrapper():
        return '<strong>' + func() + '</strong>'
    return wrapper

def emphasis(func):
    def wrapper():
        return '<em>' + func() + '</em>'
    return wrapper
```

现在，让我们使用这两个装饰器，并将它们同时应用于我们的`greet`函数。你可以使用@语法，在一个函数的顶部“堆叠”多个装饰器即可：

```
@strong
@email
def greet():
    return 'Hello!'
```

如果你运行被装饰后的函数，你希望看到什么输出？`@emphasis`装饰器将首先添加标签，还是`@strong`先执行？这是当你调用被装饰函数时发生的情况：

```
>>> greet()
'<strong><em>Hello!</em></strong>'
```

这清楚地显示了装饰器的应用顺序：从下到上。首先，输入函数由`@emphasis`装饰器装饰，然后得到的函数再次由`@strong`装饰器装饰。

为了帮助我记住这种自下而上的顺序，我喜欢将其称为行为装饰器堆栈。你从底部开始构建堆栈，然后继续在顶部添加新的块以向上移动。

如果分解上面的示例并避免使用@语法装，装饰器函数调用链如下所示：

```
decorated_greet = strong(emphasis(greet))
```

你再次看到`emphasis`装饰器首先被应用，然后，被装饰后的函数再由`strong`装饰器装饰。

深层次的装饰器不断添加嵌套函数调用，堆叠装饰器最终会影响性能。实际上，这通常不会有问题，但是如果你要编写频繁使用装饰器的性能密集型代码，则需要牢记这一点。

装饰带参数的函数

到目前为止，所有示例仅装饰了一个简单的没什么用的greet函数，这个函数没有接受任何参数。目前你在此处看到的装饰器还没有处理向输入函数传参的事情。

如果你尝试将这些装饰器之一应用于带有参数的函数，它将无法正常工作。那么如何装饰需要任意参数的函数？

这就是Python用于处理可变数量参数的 *args 和 **kwargs 特性起作用的时候。以下代理装饰器利用这个特性：

```
def proxy(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper
```

有两件事情需要注意：

它在 wrapper 闭包中使用 * 和 ** 运算符接受所有位置参数和关键字参数并将其存储在变量中（args 和 kwargs）。

然后，wrapper 闭包使用 * 和 ** 参数解包运算符，将接受到的参数转发到原始输入函数。

不幸的是，* 和 ** 运算符意义过重，并且其意义会根据上下文发生变化，但我希望你能明白这一点。

让我们将代理装饰器设计的技术扩展到更有用的实际例子。这是记录函数执行期间的参数和结果的跟踪装饰器：

```
def trace(func):
    def wrapper(*args, **kwargs):
        print(f'TRACE: calling {func.__name__}() '
              f'with {args}, {kwargs}')
        original_result = func(*args, **kwargs)
        print(f'TRACE: {func.__name__}() '
              f'returned {original_result!r}')
        return original_result
    return wrapper
```

用 trace 装饰器装饰函数，然后调用它，将打印出传递给被装饰函数的参数及其返回值。这个例子仍然有点像“玩具”，但在紧要关头，它确实可以起到很好的辅助调试的作用：

```
@trace
def say(name, line):
    return f'{name}: {line}'

>>> say('Jane', 'Hello, World')
'TRACE: calling say() with ("Jane", "Hello, World"), {}'
'TRACE: say() returned "Jane: Hello, World"'
'Jane: Hello, World'
```

说到调试，当你调试装饰器时，应该记住这些事情：

怎么书写“可调试”的装饰器

当你使用装饰器时，实际上你正在做的是用一个函数替换另一个。这么做的一个缺点是它“隐藏”了原始（未被修饰）函数的一些元数据。

例如，原始函数的名称，docstring 和参数列表被 wrapper 闭包隐藏：

```
def greet():
    """Return a friendly greeting."""
    return 'Hello!'

decorated_greet = uppercase(greet)
```

如果你尝试访问该函数元数据中的任何一个，就会看到 wrapper 闭包的元数据：

```
>>> greet.__name__
'greet'
>>> greet.__doc__
'Return a friendly greeting.'
>>> decorated_greet.__name__
'wrapper'
>>> decorated_greet.__doc__
None
```

这使得调试装饰器变得困难。幸运的是，有一个快速解决方法：Python标准库中的 `functools.wraps` 装饰器

你可以在自己的装饰器中使用 `functools.wraps` 把未修饰的函数的源数据复制到装饰器闭包。这是一个例子：

```
import functools
def uppercase(func):
    @functools.wraps(func)
    def wrapper():
        return func().upper()
    return wrapper
```

在装饰器返回的wrapper闭包上使用 `functools.wraps` 可以将输入函数的`docstring`和其他元数据携带过来：

```
@uppercase
def greet():
    """Return a friendly greeting."""
    return 'Hello!'

>>> greet.__name__
'greet'
>>> greet.__doc__
'Return a friendly greeting.'
```

最佳做法是，建议你在自己编写的所有装饰器使用 `functools.wraps`。这不需要很多时间，但是它将为你（和其他人）免去日后调试的麻烦。

哦，恭喜！你已经成功完成了这一复杂的章节，并从中学习了很多关于装饰器的知识。做得好！

重点

装饰器定义了可以修改你可调用对象的代码块，使用装饰器无需永久修改可调用对象本身。

- @语法只是调用装饰器的简写形式。单个函数上的多个装饰器是从下到上应用（装饰器堆叠）。
- 作为调试的最佳实践，请在你自己的装饰器中使用 `functools.wraps` 装饰器将元数据从未经修饰的可调用对象继承到装饰的对象。
- 与软件开发工具箱中的其他任何工具一样，装饰器不是万能的，也不应该被过度使用。重要的是平衡“完成工作”与“不要陷入可怕的，无法维持的一堆烂代码。”

3.4 使用 `*args` 和 `**kwargs` 的函数

我曾经和一个聪明的Python开发结对编程，他会在每个带有可选参数或关键字参数的函数中声明`argh`和`kwargh`。然而我们相处得很好。我想这就是学术界的编程最终对人的影响。

现在，虽然很容易绕过，但 `*args` 和 `**kwargs` 参数仍然是Python中非常有用的功能。了解他们将使你成为更高效的开发人员。

那么`* args` 和 `** kwargs` 参数的作用是什么呢？他们允许函数接受可选参数，因此你可以在你的模块和类中创建灵活的API：

```
def foo(required, *args, **kwargs):
    print(required)
    if args:
        print(args)
    if kwargs:
        print(kwargs)
```

上面的函数至少需要一个名为`required`的参数，但它也可以接受额外的位置参数和关键字参数。

如果我们使用其他参数调用该函数，因为参数名称有`*`前缀，`args`将收集多余的位置参数作为一个元组。同理，因为参数名称带有`**`前缀，`kwargs`将收集额外的关键字参数作为一个字典。

如果没有传递额外的参数，`args`和`kwargs`都可以为空。

当我们使用各种参数组合调用函数时，你将看到Python如何根据它们是位置参数还是关键字参数将它们收集在`args`和`kwargs`中：

```
>>> foo()
TypeError:
"foo() missing 1 required positional arg: 'required'""

>>> foo('hello')
hello

>>> foo('hello', 1, 2, 3)
hello
(1, 2, 3)

>>> foo('hello', 1, 2, 3, key1='value', key2=999)
hello
(1, 2, 3)
{'key1': 'value', 'key2': 999}
```

我想明确指出，参数`args`和`kwargs`只是一个命名约定。如果你将它们称为`parms`和`**argv`，前面的例子还是可以正常工作。实际的语法其实只是星号`()`或双星号`(**)`。

但是，我建议你遵守公认的命名约定，以免造成混淆。

转发可选参数和关键字参数

可以将可选参数或关键字参数从一个函数传递到另一个函数。你可以通过在调用要转发参数的函数时使用参数拆包运算符`*`和`**`来实现。

这也使你有机会在传递参数之前修改他们。这是一个例子：

```
def foo(x, *args, **kwargs):
    kwargs['name'] = 'Alice'
    new_args = args + ('extra', )
    bar(x, *new_args, **kwargs)
```

这个技术对于子类化和编写包装器功能很有用。例如，你可以使用它来扩展父类的行为而不必在子类中复制父类构造函数的完整签名。如果你正在使用在你控制范围之外的API，这可能会非常方便：

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

class AlwaysBlueCar(Car):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.color = 'blue'

>>> AlwaysBlueCar('green', 48392).color
'blue'
```

`AlwaysBlueCar`构造函数只需将所有参数传递给其父类，然后覆盖内部属性。如果父类的构造函数发生变化，`AlwaysBlueCar`很可能仍将按预期运行。

缺点是`AlwaysBlueCar`构造函数现在有一个相当无用的签名——如果不查父类，我们不知道它期望什么参数。

通常，你不会将这种技术用于你自己的类继承中。你可能更倾向修改或覆盖一些你无法控制的外部类中的行为。

但这始终是一个危险的领域，因此最好小心。

这种技术的另一种常用场景是编写像装饰器一类的包装函数。这种场景下，通常希望可以接受任意参数并将其传递给包装函数。

而且，如果我们能够做到不用复制和粘贴原始函数的签名，可能会更易于维护：

```

def trace(f):
    @functools.wraps(f)
    def decorated_function(*args, **kwargs):
        print(f, args, kwargs)
        result = f(*args, **kwargs)
        print(result)
    return decorated_function

@trace
def greet(greeting, name):
    return '{}, {}!'.format(greeting, name)

>>> greet('Hello', 'Bob')
<function greet at 0x1031c9158> ('Hello', 'Bob') {}
'Hello, Bob!'

```

使用这样的技术，有时很难在保持代码足够明确的同时，仍遵守不要重覆自己（DRY）原则。这永远是一个艰难的选择。如果你可以从同事那里获得其他意见，我建议你听听他们的意见。

重点

- 使用`*args`和`**kwargs`可以编写参数数量不确定的Python函数。
- `*args`收集额外的位置参数作为元组。`**kwargs`收集额外的关键字参数作为字典。
- 实际语法为`*`和`**`。称它们为`args`和`kwargs`只是一项约定

3.5 函数参数解包

一个非常酷但有点神秘的特性是能够使用`*`和`**`操作符从序列和字典中“解包”函数参数。

让我们定义一个简单的函数作为示例：

```

def print_vector(x, y, z):
    print('<%s, %s, %s>' % (x, y, z))

```

你看，这个函数接收三个参数（`x`, `y`和`z`），并且用一种整洁的格式打印它们。我们可能会在我们程序中使用此函数打印3维向量：

```

>>> print_vector(0, 1, 0)
<0, 1, 0>

```

现在，根据我们选择描述3维向量的数据结构的不同，使用我们的`print_vector`函数打印它们会有点蹩脚。例如，如果我们的向量表示为元组或列表，打印它们时我们必须为每个组件明确指定索引：

```

>>> tuple_vec = (1, 0, 1)
>>> list_vec = [1, 0, 1]
>>> print_vector(tuple_vec[0],
tuple_vec[1],
tuple_vec[2])
<1, 0, 1>

```

使用带有单独参数的普通函数调用似乎显得冗长且繁琐。如果可以将一个矢量对象“分解”为其三个组成部分，并将所有东西一次全部传递给`print_vector`函数，岂不更好？

（当然，你可以简单地重新定义`print_vector`，以使它接收代表矢量对象的单个参数——但为了举一个简单的例子，我们现在先忽略该选项。）

值得庆幸的是，有一种更好的方法可以解决Python中的这种情况：使用`*`运算符的函数参数解包：

```

>>> print_vector(*tuple_vec)
<1, 0, 1>
>>> print_vector(*list_vec)
<1, 0, 1>

```

在函数调用中将*放在可迭代对象之前将对可迭代对象进行解包，并将其元素作为单独的位置参数传递给调用函数。

该技术适用于任何可迭代对象，包括生成器表达式。在生成器上使用*运算符会从生成器取出所有元素传递给函数：

```
>>> genexpr = (x * x for x in range(3))
>>> print_vector(*genexpr)
```

除了用于解包像元组、列表和生成器这一类序列并将其转换为位置参数的*运算符，还有用于从字典中解包关键字参数的**运算符。假设我们的向量用以下字典表示：

```
>>> dict_vec = {'y': 0, 'z': 1, 'x': 1}
```

我们可以使用**运算符以相似的方式将这个字典传递给print_vector函数：

```
>>> print_vector(**dict_vec)
<1, 0, 1>
```

由于字典是无序的，因此这里基于字典键对函数参数和对应的值进行匹配：x参数接收与字典中的“x”键对应的值。

如果要使用单星号(*)运算符对字典进行解包，则使用随机顺序将键传递给函数：

```
>>> print_vector(*dict_vec)
<y, x, z>
```

Python的函数参数解包功能为你提供了很多灵活性。通常，这意味着你不必实现一个程序所需的数据类型的类。使用像元组或列表这样简单的内置数据结构就足够了，并有助于减少代码的复杂度。

重点

- *和**运算符可用于从序列和字典中“解包”函数参数。
- 有效使用参数解包可以帮助你编写更灵活的接口。

3.6 没有返回值

Python在任何函数的末尾隐式添加了return None语句。因此，如果函数未指定返回值，则默认情况下它将返回None。

这意味着你可以将return None语句替换为return语句，甚至将它们完全丢弃，仍然会得到相同的结果：

```
def foo1(value):
    if value:
        return value
    else:
        return None

def foo2(value):
    """Bare return statement implies return None"""
    if value:
        return value
    else:
        return

def foo3(value):
    """Missing return statement implies return None"""
    if value:
        return value
```

如果你给这三个函数都传递假的值作为其唯一参数，则这三个函数都将正确返回None：

```
>>> type(foo1(0))
<class 'NoneType'>

>>> type(foo2(0))
```

```
<class 'NoneType'>  
>>> type(foo3(0))  
<class 'NoneType'>
```

那么什么时候使用这个功能呢？

我的经验是，如果一个函数没有返回值（其他语言将其称为过程），那么我将省略return语句。这种情境下，添加一个return语句显得有点多余和混乱。一个过程的示例是Python内置的print函数，它因其副作用（打印文本）而被调用，而从不使用其返回值。

让我们看看Python内置的sum函数。很明显，它具有逻辑返回值，通常sum不会仅因其副作用而被调用。其目的是将一系列数字加在一起，然后返回结果。现在，如果从逻辑的角度来看某个函数的确具有返回值，那么你需要确定是否使用隐式返回。

一方面，你可能会争辩说省略了明确的return None语句使代码更简洁，因此更容易阅读和理解。主观上，你可能还说它使代码“更漂亮”。

另一方面，某些程序员可能会对Python这样的行为感到惊讶。当谈到干净和可维护代码时，令人惊讶的行为不是一个好兆头。

例如，我在本书的早期版本中的代码示例中使用了“隐式return语句”。我没提到我在做什么——我只想要一个不错的短代码示例来解释Python中的一些其他功能。

最终，我开始收到源源不断的电子邮件，该代码示例中的“缺少的return语句”。显然Python的隐式返回行为不是对每个人都很明显，在这种情况下会分散注意力。我添加了一条便条以说明正在发生的事情，接着，电子邮件停止了。

不要误会我的意思——我和其他人一样喜欢写简洁而“美丽”的代码。而且我以前也强烈认为程序员应该了解他们使用的语言的来龙去脉。

但是当你考虑到即使如此简单的误会所带来的可维护性影响时，多写几行代码，让代码更加清晰就更加有意义了。毕竟，代码就是沟通。

重点

- 如果函数未指定返回值，则返回None。是否显式返回None与风格相关。
- 这是Python的核心功能，但是使用显式的return None语句，可以让你的代码更清晰。

source/python_tricks/4_类和面向对象.md

第四章 类和面向对象

4.1 对象比较，is还是==

小时候，我的邻居有一对双胞胎猫。它们看起来似乎完全相同——相同的木炭色皮毛和相同的绿色瞳孔。除了一些古怪的个性，你无法通过观察它们将它们分开。当然，虽然它们看起来完全一样，但是它们是两只不同的猫。

这使我想到了相等与相同之间的意义差异。而这种差异对于理解Python中 is 和 == 比较运算符的行为至关重要。

== 运算符通过检查是否相等来进行比较：如果这些猫是Python对象，将它们用 == 运算符进行比较，我们将得到“两只猫相等”的结论。

但是，is运算符会比较特性：如果我们使用is运算符，我们会得到“这两只猫不同”的结论。

但是，在我们陷入这个问题之前，让我们看一些真实的例子。首先，我们创建一个新的列表对象并将其命名为a，然后定义另一个变量b，将其指向同一个列表对象：

```
>>> a = [1, 2, 3]
>>> b = a
```

让我们检查一下这两个变量。我们可以看到他们指向看起来相同的列表：

由于两个列表对象看起来相同，因此当我们使用 == 运算符比较它们时，我们可以得到预期的结果：

```
>>> a == b
True
```

但是，这并不能告诉我们a和b是否实际上指向相同的对象。当然，我们知道它们是，因为我们早些时候为它们进行了赋值，但假设我们不知道，我们该怎么确定呢？

答案是使用 is 运算符比较两个变量。这样可以确认两个变量实际上都指向一个列表对象：

```
>>> a is b
True
```

让我们看看当我们创建列表的副本时会发生什么。我们可以通过在现有列表上调用 list() 来创建副本，我们将其命名为c：

```
>>> c = list(a)
```

同样，你会看到我们刚刚创建的新列表看起来与a和b指向的列表是相同的：

```
>>> c
[1, 2, 3]
```

现在，事情开始变得有趣了。让我们使用 == 运算符对初始列表和其副本c进行比较。你期望得到什么答案？

```
>>> a == c
True
```

好的，我希望这就是你期望的结果。这个结果告诉我们列表c和a具有相同的内容。在Python中它们被认为是相等的。但是它们实际上是指向同一个对象吗？让我们使用 is 运算符看看：

```
>>> a is c
False
```

Boom! 这里我们得到了不同的结果。c和a指向两个不同的对象，即使它们内容可能相同。

回顾一下，让我们尝试用两个简短的定义区分 `is` 和 `==`：

- 如果两个变量指向相同的对象，`is` 表达式结果为True。
- 如果变量引用的对象相等，`==` 表达式的结果为True。

你需要决定使用`is`和`==`之前，想一想双胞胎猫（狗也行）。

4.2 字符串转换（每个类都需要一个 `__repr__` 方法）

当你在Python中定义自己的类，然后尝试在控制台打印它的一个实例时（或在解释器会话中检查它），你会得到一个不那么令人满意的结

果。默认的字符串转换行为是很基础的，并且缺乏必要的细节：

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

>>> my_car = Car('red', 37281)
>>> print(my_car)
<__console__.Car object at 0x109b73da0>
>>> my_car
<__console__.Car object at 0x109b73da0>
```

默认情况下，你得到的只是一个包含类名称和ID的字符串（这是CPython中对象的内存地址）。有总比没有好，但就是有点鸡肋。

你可能会尝试直接打印类的属性，甚至通过在类中添加自定义的 `to_string()` 来解决此问题。

```
>>> print(my_car.color, my_car.mileage)
red 37281
```

大致的想法是正确的，但它忽略了Python将对象表示为字符串的约定和内置机制。

你无需构建自己的字符串转换机制，最好是在类中添加 `__str__` 和 `__repr__` 方法。它们是控制在不同情况下将对象转换为字符串的方法，这种方式更加符合python的风格。让我们看一下这些方法在实际中的应用。我们先在类Car的定义中添加 `__str__` 方法：

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __str__(self):
        return f'a {self.color} car'
```

现在，当你尝试打印或检查Car的实例时，将会看到一个和之前不同，略有改善的结果：

```
>>> my_car = Car('red', 37281)
>>> print(my_car)
'a red car'
>>> my_car
<__console__.Car object at 0x109ca24e0>
```

在控制台中检查Car对象仍会输出包含对象ID的结果，这和之前没有区别。但是打印对象会得到我们添加的 `__str__` 方法返回的字符串。

`__str__` 是Python的双下划线方法之一，会在你尝试将对象转换为字符串时被调用：

```
>>> print(my_car)
a red car

>>> str(my_car)
'a red car'
>>> '{}'.format(my_car)
'a red car'
```

有了正确的`__str__`实现后，你就不必再关心直接打印对象属性或编写单独的`to_string()`函数。这是Pythonic的控制字符串转换的方式。

顺便说一句，有些人将Python的双下划线方法称为“魔术方法”。但是这些方法一点也不魔幻。这些方法以双下划线开始和结束只是Python将其标记为核心特性的命名约定。它还有助于避免与你自己的方法和属性产生命名冲突。对象初始化函数`__init__`遵循同样的约定，没有什么神奇或神秘的东西。

不要害怕使用Python的双下划线方法——它们实际上是为了帮助你。

`__str__` vs `__repr__`

现在，我们关于字符串转换的故事还没有结束。你看到了吗？在解释器中检查`my_car`对象时，仍然输出了奇怪的`<Car object at 0x109ca24e0>`？

这是因为在Python 3中实际上有两种下划线方法在控制如何将对象转换为字符串。第一个是你刚刚了解的`__str__`。第二个是`__repr__`，它的工作方式类似于`__str__`，但是它用在不同的场景。（Python 2.x还有一个`__unicode__`方法，我会在稍后介绍）。

这是一个简单的例子，你可以用来自了解什么时候使用`__str__`，什么时候使用`__repr__`。让我们重新定义我们的`Car`类，让它包含`__str__`和`__repr__`方法，同时又具有易于区分的输出：

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __repr__(self):
        return '__repr__ for Car'

    def __str__(self):
        return '__str__ for Car'
```

现在，当你浏览前面的示例时，你可以看到在每种情况下哪种方法在控制字符串的转换：

```
>>> my_car = Car('red', 37281)
>>> print(my_car)
__str__ for Car
>>> '{}'.format(my_car)
'__str__ for Car'
>>> my_car
__repr__ for Car
```

此实验证实，在Python解释器会话中检查对象只是会打印对象的`__repr__`方法输出的结果。有趣的是，诸如列表和字典之类的容器始终使用`__repr__`代表它们包含的对象。即使你在容器本身上调用`__str__`方法：

```
str([my_car])
['__repr__ for Car']
```

要在这两种字符串转换方法之间进行选择以更清楚地表达代码的意图时，最好使用内置的`str()`和`repr()`函数。与直接调用对象的`__str__`或`__repr__`属性相比，使用它们更可取，因为它更好看，并且输出相同的结果：

```
>>> str(my_car)
 '__str__ for Car'
>>> repr(my_car)
 '__repr__ for Car'
```

即使这项研究结束了，你可能仍想知道`__str__`和`__repr__`之间的差异是什么？他们两个似乎都达到了相同的目的，因此什么时候使用那个还不是很清楚。

遇到诸如此类的问题，通常最好先查看一下Python标准库的做法。是时候设计另一个实验了。我们将创建一个`datetime.date`对象，观察它怎么使用`__repr__`和`__str__`来控制字符串转换：

```
>>> import datetime
>>> today = datetime.date.today()
```

日期对象的 `__str__` 函数的结果首先要是可读的。这意味着要返回简洁明了的文字形式以供人阅读——以你觉得很舒服的方式显示给用户。

因此，当我们在日期对象上调用 `str()` 时，得到的东西看起来像是ISO日期格式：

```
>>> str(today)
'2017-02-02'
```

使用 `__repr__` 最重要的理念是其结果应该是明确的。产生的字符串更多地被开发人员用作调试的辅助工具。为此，它需要尽可能明确表述这个对象是什么。这就是为什么你在对象上调用 `repr()` 会得到更详尽结果的原因。它甚至包括完整的模块和类名称：

```
>>> repr(today)
'datetime.date(2017, 2, 2)'
```

我们可以复制并粘贴 `__repr__` 返回的字符串并将其作为有效的Python代码执行来重新创建原始的 `date` 对象。这是编写自己的 `repr` 方法时要牢记的一个目标。

另一方面，我发现这很难付诸实践。通常情况下，这样做是不值得的，它只会成为你额外的负担。我的经验是使 `__repr__` 字符串清晰无误并且对开发人员有帮助，但我不希望他们能够还原对象的完整状态。

为什么每个类都需要 `__repr__`

如果你不添加 `__str__` 方法，在调用 `__str__` 时会返回 `__repr__` 的结果。因此，我建议你至少在类中添加 `__repr__` 方法。这会在最少的投入下保证几乎在所有情况下都有有用的字符串转换结果。

以下是快速有效地向你的类添加基本字符串转换支持的方法。对于我们的Car类，我们可以从以下 `__repr__` 开始：

```
def __repr__(self):
    return f'Car({self.color!r}, {self.mileage!r})'
```

请注意我正在使用 `!r` 转换标志来确保输出字符串使用 `repr(self.color)` 和 `repr(self.mileage)` 而不是 `str(self.color)` 和 `str(self.mileage)` 的值。

这样效果很好，但缺点是在格式字符串中我们重复了类的名称。一个可以在此处避免这种重复的技巧是使用对象的 `__class__.__name__` 属性，该属性始终将类名称转换为字符串。

好处是当类名更改时你无需修改 `__repr__` 实现。这样可以轻松遵守不要重复自己（DRY）的原则：

```
def __repr__(self):
    return (f'{self.__class__.__name__}({self.color!r}, {self.mileage!r}))'
```

该实现的缺点是格式字符串相当笨拙冗长。但是通过仔细的格式化，你可以保持代码整洁且符合PEP 8规范。

通过上面的 `__repr__` 实现，当我们检查对象或直接在其上调用 `repr()` 时，我们获得了一个有用的输出：

```
>>> repr(my_car)
'Car(red, 37281)'
```

打印对象或在其上调用 `str()` 会返回相同的字符串，因为默认的 `__str__` 实现只是简单调用 `__repr__`：

```
>>> print(my_car)
'Car(red, 37281)'
>>> str(my_car)
'Car(red, 37281)'
```

我认为这种方法在提供最大的价值的同时确保了不需要过多的代码。这也是一种千篇一律的方法，无需过多考虑就可以应用。因此，我总是尝试将基本的 `__repr__` 实现添加到我的类中。

这是Python 3的完整示例，其中包括一个可选的 `__str__` 实现：

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage
    def __repr__(self):
        return f'{self.__class__.__name__}({self.color!r}, {self.mileage!r})'
    def __str__(self):
        return f'a {self.color} car'
```

Python2 的不同: `__unicode__`

在Python 3中，`str` 可以全面表示文本。它包含 `unicode` 字符，可以代表世界上大多数国家/地区的书写系统。

Python 2.x对字符串使用了不同的数据模型。有两个表示文本的类型：`str` 和 `unicode`，其中 `str` 仅限于ASCII字符集，`unicode`，等效于Python 3的`str`。

由于存在这种差异，因此在Python 2中有另一个用于控制字符串转换的双下划线方法：`__unicode__`。在Python 2中，`__str__` 返回字节，而`__unicode__` 返回字符。

大多数情况下，`__unicode__` 是控制字符串转换的新和首选的方法。还有一个内置的 `unicode()` 函数与之一起使用。它调用各自的双下划线方法，这与 `str()` 和 `repr()` 类似。

到现在为止还挺好的。现在，当你查看在Python 2中调用 `__str__` 和 `__unicode__` 的规则时，事情就有点复杂了：`print`语句和 `str()` 调用 `__str__`。如果 `__unicode__` 存在，内置的 `unicode()` 调用 `__unicode__`，否则回退到 `__str__` 并使用系统文本编码对结果进行解码。

与Python 3相比，这些特殊情况使文本转换规则更加复杂。但是实际使用中有一种方法可以对其进行简化。在Python程序中处理文本时，`unicode` 是首选的方法。

因此，一般而言，我建议你在Python 2.x把所有字符串格式代码放到 `__unicode__` 方法中，然后在 `__str__` 中返回UTF-8编码的 `unicode` 表示：

```
def __str__(self):
    return unicode(self).encode('utf-8')
```

对于你编写的大多数类而言，`__str__` 方法都是相同的，因此你可以根据需要进行复制和粘贴（或将其放入基类中）。所有你打算给非开发人员使用的字符串转换代码都放在 `__unicode__` 中。

这是Python 2.x的完整示例：

```
class Car(object):
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __repr__(self):
        return '{}({!r}, {!r})'.format(self.__class__.__name__, self.color, self.mileage)

    def __unicode__(self):
        return u'a {} car'.format(self=self)

    def __str__(self):
        return unicode(self).encode('utf-8')
```

重点

- 你可以在自己的类中使用 `__str__` 和 `__repr__` 双下划线方法控制字符串转换。
- `__str__` 的结果应可读，`__repr__` 的结果应该是明确的。
- 始终在类中添加 `__repr__`，`__str__` 的默认实现是调用 `__repr__`。
- 在Python 2中使用 `__unicode__` 代替 `__str__`。

4.3 定义你自己的异常类

当我开始使用Python时，我犹豫要不要在我的代码中编写自定义异常类。但是定义自己的错误类型可能有用。你可以清楚地指出潜在的错误场景，并且你的函数和模块将变得更加易于维护。你还可以使用自定义的错误类型来提供其他调试信息。

所有这些都将改善你的Python代码，使其更易于理解，调试和维护。当你将其分解为几个简单的类别时，定义自己的异常类别并不难。在本章中，我将引导你了解需要记住的要点。

假设你要在应用程序中验证代表人名的输入字符串。名称验证器的简单示例可能如下所示：

```
def validate(name):
    if len(name) < 10:
        raise ValueError
```

如果验证失败，将抛出ValueError异常。那看起来还算合适，并且也有点Pythonic。到现在为止还挺好。

但是，使用像ValueError这样的“高级”通用异常是不利的。想象一下，你的一个对其内部不是很了解的同事在一个包中调用了这个函数。当名称无法通过验证时，错误栈看起来像这样：

```
>>> validate('joe')
Traceback (most recent call last):
File "<input>", line 1, in <module>
validate('joe')
File "<input>", line 3, in validate
raise ValueError
ValueError
```

这个错误栈并不是那么有用。当然，我们知道某些地方出了问题，并且问题与“值不正确”有关。但你的队友为了解决这个问题，几乎必须查看 validate() 的实现。然而，阅读代码会花费时间。而且它可以快速累加。

幸运的是我们可以做得更好。让我们介绍一个表示名称验证失败的自定义异常类型。我们将在Python内置的ValueError基础上构建新的异常类，通过给它一个更明确的名称，可以让它自解释：

```
class NameTooShortError(ValueError):
    pass

def validate(name):
    if len(name) < 10:
        raise NameTooShortError(name)
```

现在，我们有了一个“自解释”的 NameTooShortError 异常类型，它通过扩展内置 ValueError 类而来。通常，你会从根 Exception 或其他内置的Python异常（如 ValueError 或 TypeError 或其它合适的类）派生自定义异常。

另外，看看我们现在如何在 validate 中实例化自定义异常类时将 name 变量传递给其构造函数。新的实现有更好的错误栈信息：

```
>>> validate('jane')
Traceback (most recent call last):
File "<input>", line 1, in <module>
validate('jane')
File "<input>", line 3, in validate
raise NameTooShortError(name)
NameTooShortError: jane
```

再一次尝试换位思考。当事情出错时（最终他们总是会出错），自定义异常可让你更轻松地了解正在发生的事情。

即使你自己在一个人编写代码，情况也是如此。如果代码结构合理，在接下来的几周或几个月内，你维护你的代码时将更轻松。

通过仅花费30秒来定义一个简单的异常类，这代码段已经变得更清晰。但是，让我们继续前进。还有更多内容要讲。

每当你公开发布Python软件包，甚至为你的公司创建可重用的模块时，最好的做法是为该模块创建一个自定义的异常基类，然后派生所有其他异常。

以下是在模块或包装中为所有异常创建自定义异常层次结构的方法。第一步是声明一个我们所有的异常都将继承的基类：

```
class BaseValidationError(ValueError):
    pass
```

现在，我们所有的“真实”异常类都可以从 `Base` 异常类中继承。几乎不用多做什么，这给出了一个简洁的异常层次结构，：

```
class NameTooShortError(BaseValidationError):
    pass

class NameTooLongError(BaseValidationError):
    pass

class NameTooCuteError(BaseValidationError):
    pass
```

例如，这允许使用你包的用户在使用 `try ... except` 语句时，可以处理此包中所有错误的语句，而无需单独手动捕获它们：

```
try:
    validate(name)
except BaseValidationError as err:
    handle_validation_error(err)
```

人们仍然可以通过这种方式捕获更具体的异常，但是如果他们不想，至少他们不必捕获所有异常。这通常被认为是一种反模式——它可以默默地吞噬并隐藏不相关的错误并使你的程序难以调试。

当然，你可以将这个想法更进一步，并在逻辑上将你的想法分为细粒度的子层次结构。但请注意，过度设计很容易引入不必要的复杂性。

总而言之，定义自定义异常类可以让你的用户更容易使用宽恕比许可要容易（EAFP）的编码风格，这被认为是更Pythonic的。

重点

- 定义自己的异常类可以使代码的意图更清晰并易于调试
- 从Python的内置异常衍生自定义异常类或更具体的异常类如`ValueError`或`KeyError`。
- 你可以使用继承来实现逻辑分组的异常层次结构。

4.4 克隆对象的乐趣和收益

Python中的赋值语句不会创建对象的副本，它们是将名称绑定到对象。对于不可变的对象也是一样。

但是对于可变对象或可变对象的集合，你可能在寻找一种方法来创建其“真实副本”或“克隆”这些对象。

本质上，你有时会想要可以修改而无需同时自动修改原始数据。在这一章当中我将向你介绍在Python中如何复制或“克隆”对象，以及一些注意事项。

首先，我们来看看如何复制Python的集合。Python内置的可变集合（例如列表，字典和集合）可以通过在现有集合上调用其工厂函数来复制：

```
new_list = list(original_list)
new_dict = dict(original_dict)
new_set = set(original_set)
```

但是，此方法不适用于自定义对象，此外，那只是浅拷贝。对于列表、字典和集合等复合对象，浅拷贝和深拷贝之间有一个重要的区别：

浅拷贝意味着构造一个新的集合对象，然后使用原始对象中子对象的引用来填充它。本质上，浅拷贝的副本仅深一层。复制过程不会递归，因此不会创建子对象自己的副本。

深拷贝的复制过程是递归的。这意味着首先构造一个新的集合对象，然后使用在原始文档中找到的子对象的副本递归地填充它。以这种方式复制对象将遍历整个对象树以创建与原始对象及其所有子对象完全独立的副本。

我知道，这有点绕。因此，让我们看一些例子来了解深拷贝和浅拷贝之间的差异。

创建浅拷贝

在下面的示例中，我们将创建一个新的嵌套列表，然后使用 `list()` 工厂函数进行浅拷贝：

```
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys = list(xs) # Make a shallow copy
```

这意味着 `ys` 现在是一个与 `xs` 拥有相同内容的独立对象。可以通过检查两个对象来验证这一点：

```
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

为了确认 `ys` 确实与原始列表无关，让我们设计一个小实验。你可以尝试将新的子列表添加到原始列表中（`xs`），然后检查以确保此修改不会影响复制的列表（`ys`）：

```
>>> xs.append(['new sublist'])
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9], ['new sublist']]
>>> ys
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

你看，这产生了预期的效果。在“表层”修改复制列表根本没有问题。

但是，由于我们仅创建了原始列表的浅拷贝，`ys` 仍然包含对 `xs` 中存储的原始子对象的引用。

这些子对象并没有被复制，在复制的列表中它们只是再次被引用。

因此，当你修改 `xs` 中的一个子对象时，此修改也将反映在 `ys` 中——这是因为两个列表共享相同的子对象。该副本只是一个一层深度的复制：

```
>>> xs[1][0] = 'X'
>>> xs
[[1, 2, 3], ['X', 5, 6], [7, 8, 9], ['new sublist']]
>>> ys
[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]
```

在上面的示例中，我们看上去仅对 `xs` 进行了更改。但实际上，在 `xs` 和 `ys` 中索引 1 的两个子列表都被修改了。同样，发生这种情况是因为我们仅创建了一个原始列表的浅拷贝。

如果我们在第一步中创建了 `xs` 的深拷贝副本，那么这两个对象将是完全独立的。这是浅拷贝和深拷贝的实际区别。

现在你知道了如何创建一些内置容器类的浅拷贝，并且你知道了浅拷贝和深拷贝之间的区别。我们仍然需要回答的问题是：

- 如何创建内置集合的深拷贝？
- 如何创建包括自定义类在内的任意对象的副本（浅拷贝和深拷贝）？

这些问题的答案都在 Python 标准库的 `copy` 模块中。该模块提供了一个用于创建任意 Python 对象的浅拷贝和深拷贝的简单接口。

创建深拷贝

让我们重复前面的列表复制示例，但是有一个重要的区别。这次，我们将使用在复制模块中定义的 `deepcopy()` 函数来实现：

```
>>> import copy
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> zs = copy.deepcopy(xs)
```

当你检查 `xs` 及使用 `copy.deepcopy()` 创建的克隆 `zs` 时，就像前面的例子一样，它们看起来是相同的：

```
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> zs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

但是，如果你修改了原始对象（xs），你会看到此修改不会影响深拷贝（zs）。

这一次，原始对象和副本是完全独立的。xs和其所有子对象都是递归克隆的：

```
>>> xs[1][0] = 'X'
>>> xs
[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]
>>> zs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

你可能需要花一些时间在Python解释器上把玩一下这些示例。当你亲身体验一些示例并有一定经验时，理解拷贝就会更轻松。

顺便说一句，你也可以使用复制模块创建浅拷贝。`copy.copy()` 函数创建对象的浅拷贝。

如果你需要在代码中的某处清楚地说明自己正在创建一个浅拷贝。使用 `copy.copy()` 可让你说明这一事实。但是，对于内置集合，列表，字典，使用工厂函数来创建浅拷贝是一种更Pythonic的方法。

复制任意对象

我们仍然需要回答的问题是如何创建包括自定义类在内的任意对象的副本（深拷贝和浅拷贝）。让我们现在来看一看。

我们可以使用 `copy` 模块解决这个问题。它的 `copy.copy()` 和 `copy.deepcopy()` 函数可用于复制任何对象。

了解如何使用它们的最佳方法是使用一个简单的实验。我将在先前的列表复制例子上继续，首先定义一个简单的2D点类：

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Point({self.x!r}, {self.y!r})'
```

我希望你同意这很简单。我加了一个 `__repr__()` 实现，以便我们可以在Python解释器中轻松检查从此类创建的对象。

接下来，我们将创建一个 `Point` 实例，然后使用复制模块对其进行浅拷贝：

```
>>> a = Point(23, 42)
>>> b = copy.copy(a)
```

如果我们检查原始 `Point` 对象及其浅拷贝的内容，我们看到了我们期望的结果：

```
>>> a
Point(23, 42)
>>> b
Point(23, 42)
>>> a is b
False
```

还有其他需要注意的地方。因为我们的点对象坐标使用的是基本类型（int），在这种情况下，浅拷贝和深拷贝两者之间没有区别。但我将在下面扩展这个例子。

让我们继续一个更复杂的示例。我将定义另一个类来表示2D矩形。我将创建更复杂的对象层次结构——我的矩形会使用`Point`对象代表其坐标：

```
class Rectangle:
    def __init__(self, topleft, bottomright):
        self.topleft = topleft
        self.bottomright = bottomright

    def __repr__(self):
        return (f'Rectangle({self.topleft!r}, '
               f'{self.bottomright!r})')
```

首先，我们尝试创建一个矩形实例的拷贝：

```
rect = Rectangle(Point(0, 1), Point(5, 6))
srect = copy.copy(rect)
```

如果你检查原始矩形对象及其副本对象，你会发现效果非常好。 `__repr__()` 重载正常输出，并且浅拷贝按预期工作：

```
>>> rect
Rectangle(Point(0, 1), Point(5, 6))
>>> srect
Rectangle(Point(0, 1), Point(5, 6))
>>> rect is srect
False
```

还记得前面列表的示例怎么说明深拷贝和浅拷贝之间的差异？我将使用相同的方法。我将在对象层次结构的更深处修改一个对象，然后你会在浅拷贝中看到相应的更改：

```
>>> rect.topleft.x = 999
>>> rect
Rectangle(Point(999, 1), Point(5, 6))
>>> srect
Rectangle(Point(999, 1), Point(5, 6))
```

我希望这和你期望的一样。接下来，我将创建一个原始矩形的深拷贝。然后我将应用另一种修改，你会看到哪些对象受到了影响：

```
>>> drect = copy.deepcopy(srect)
>>> drect.topleft.x = 222
>>> drect
Rectangle(Point(222, 1), Point(5, 6))
>>> rect
Rectangle(Point(999, 1), Point(5, 6))
>>> srect
Rectangle(Point(999, 1), Point(5, 6))
```

瞧！这次，深拷贝副本完全独立于原始对象和浅拷贝对象。

我们在这里覆盖了很多领域，但关于复制对象还有一些更好的点。

深入了解该主题很有意义，因此你可能需要继续学习[复制模块文档](#)

例如，通过定义 `__copy__()` 和 `__deepcopy__()` 方法，可以控制如何复制对象。玩得开心！

重点

- 制作对象的浅拷贝并不会克隆子对象。因此，副本并不完全独立于原始对象。
- 对象的深拷贝将递归克隆子对象。完全独立于原始版本，但创建一个深拷贝速度较慢。
- 你可以使用复制模块复制包括自定义类在内的任意对象。

4.5 确保继承检查的抽象类

抽象基类（ABC）确保派生类实现基类中的特定方法。在本章中，你将学习关于抽象基类的好处以及如何使用Python内置的abc模块定义它们。

那么，抽象基类有什么用处呢？不久前，我在工作中有一次关于在Python中如何实现可维护的类继承层次结构模式的讨论。更具体地说，目标是以程序员友好和可维护的方式为服务后端定义一个简单的类层次结构。

我们有一个 `BaseService` 类，它定义了一个公共接口和一些具体的实现。具体的实现会做不同的事情，但是所有实现都提供相同的接口（`MockService`，`RealService`等）。为了使这种关系明确，所有子类具体实现都继承 `BaseService`。

为了使此代码尽可能具有可维护性和程序员友好性，我们希望确保：

- 实例化基类是不可能的；

- 忘记在子类之中实现接口方法会尽早引发异常。

现在，为什么要使用Python的 abc 模块来解决此问题？上面的设计在更复杂的系统中很常见。为了强制派生类实现基类中的许多方法，类似的Python习惯经常被使用：

```
class Base:
    def foo(self):
        raise NotImplementedError()

    def bar(self):
        raise NotImplementedError()

class Concrete(Base):
    def foo(self):
        return 'foo() called'

# Oh no, we forgot to override bar()...
# def bar(self):
#     return "bar() called"
```

那么，从第一次解决问题的尝试中我们能得到什么呢？调用Base实例上的方法正确地抛出了 `NotImplementedError` 异常：

```
>>> b = Base()
>>> b.foo()
NotImplementedError
```

此外，实例化和 `Concrete` 的使用也符合预期。而且，如果我们在其上调用诸如 `bar()` 之类没有实现的方法，也会抛出一个异常：

```
>>> c = Concrete()
>>> c.foo()
'foo() called'
>>> c.bar()
NotImplementedError
```

第一个实现不错，但还不完美。缺点是：

- 实例化 `Base` 没有抛出异常；
- 提供了不完整的子类——在我们调用缺少的方法 `bar()` 之前，实例化 `Concrete` 不会抛出异常

使用Python 2.6中添加的 abc 模块，我们可以更好地解决这些剩余的问题。这是使用 abc 模块定义的抽象基类的新实现：

```
from abc import ABCMeta, abstractmethod

class Base(metaclass=ABCMeta):
    @abstractmethod
    def foo(self):
        pass

    @abstractmethod
    def bar(self):
        pass

class Concrete(Base):
    def foo(self):
        pass
    # We forget to declare bar() again...
```

这仍然和预期一样并创建了正确的类层次结构

```
assert issubclass(Concrete, Base)
```

但是，我们确实在这里获得了另一个非常有用的好处。在我们忘记实现抽象方法时，都会在实例化子类时引发`TypeError`。引发的异常告诉我们缺少的一种或多种方法：

```
>>> c = Concrete()
TypeError:
"Can't instantiate abstract class Concrete
with abstract methods bar"
```

如果没有 abc，在缺失的方法被调用的时候，我们只会收到 `NotImplementedError`。在实例化时收到有关缺少方法的通知是一个很大的优势。这使得它更难编写无效的子类。如果你正在写新代码，这可能没什么大不了的，但几周或几个月后，我保证它将非常有用。

当然，此模式不能完全替代编译时类型检查。但是，我发现它常常使我的类层次结构更加健壮且易于维护。使用 ABC 可以清楚地表明程序员的意图，从而使代码更易于交流。我鼓励你阅读 `abc` 模块文档，密切注意应用此模式有用的情况。

重点

- 抽象基类（ABC）确保派生类在实例化时实现特定方法。
- 使用 ABC 可以帮助避免错误，并使类层次结构更易于维护。

4.6 具名元组的好处

Python带有专门的 `namedtuple` 容器类型，该容器类型似乎并没有得到应有的关注。这是那些隐藏在Python中的惊人的特性之一。

与手动定义类相比，命名元组是一种很好的选择，他们还有一些其他有趣的功能，我想在本章中为你介绍一下。

现在，什么是命名元组，它为何如此特别？一个理解 `namedtuple` 的好方法就是将它们视为内置元组的扩展类型。

Python的元组是用于将任意对象组合起来的简单数据结构。元组也是不可变的——它们一旦创建就不能修改。这是一个简短的示例：

```
>>> tup = ('hello', object(), 42)
>>> tup
('hello', <object object at 0x105e76b70>, 42)
>>> tup[2]
42
>>> tup[2] = 23
TypeError:
"'tuple' object does not support item assignment"
```

普通元组的一个缺点是你存储在其中的数据只能通过整数索引访问。你不能为存储在元组中的各个属性命名。这可能会影响代码可读性。

同样，元组始终是临时结构。很难确保两个元组具有相同数量的字段和相同的属性。这样容易因为混淆字段顺序引入bug。

可以使用具名元组解决这些问题

命名元组旨在解决这两个问题。

首先，和常规的元组一样，`namedtuple` 是不可变的容器。一旦将数据存储在 `namedtuple` 的顶级属性中后，你将无法对其进行修改。`namedtuple` 上所有属性遵循“一次写入，多次读取”的原则。

除此之外，命名元组是具有名称的元组。每个对象可以通过人类可读的唯一的标识符访问。这使你不必记住整数索引，或采用变通方法，例如将整数常量定义为用于索引的助记符。

这是一个命名元组的例子：

```
>>> from collections import namedtuple
>>> Car = namedtuple('Car', 'color mileage')
```

`namedtuple` 在Python 2.6中被添加到标准库。为了使用它们，你需要导入 `collections` 模块。在上面的例子中，我定义了一个简单的 `Car` 数据类型，其中包含两个字段：`color` 和 `mileage`。

你可能想知道在这个示例中为什么我将字符串“Car”作为第一个参数传给`namedtuple`的工厂函数。

在Python文档中，这个参数被称为“类型名”。这是通过调用`namedtuple`函数创建的新类的名称。

由于`namedtuple`无法知道我们要将结果类分配给哪个变量，因此我们需要明确地告诉它要使用哪个类名。类名用于 `namedtuple` 自动为我们生成的`docstring`和 `__repr__` 中。

在这个例子中还有另一个怪异的语法——为什么我们将字段用编码为color和mileage的字符串传递？

答案是 namedtuple 的工厂函数会在字段名称字符串上调用 `split()` 函数，将其解析为字段名称列表。所以这只是以下两个步骤的简写：

```
>>> 'color mileage'.split()
['color', 'mileage']
>>> Car = namedtuple('Car', ['color', 'mileage'])
```

当然，你也可以直接传递带有字符串字段名称的列表。使用列表的优点是你可以将其拆分为多行，可以更轻松地重新调整代码的格式：

```
>>> Car = namedtuple('Car', [
...     'color',
...     'mileage',
... ])
```

无论怎么决定，现在都可以使用 `Car` 工厂函数创建一个新的 `car` 对象。它的行为就像你手动定义了 `Car` 类并为它提供了一个接受 `color` 和 `mileage` 的构造函数一样：

```
>>> my_car = Car('red', 3812.4)
>>> my_car.color
'red'
>>> my_car.mileage
3812.4
```

除了通过其标识符访问存储在 `namedtuple` 中的值之外，你仍然可以通过其索引访问它们。这样，具名元组可以用作常规元组的替代品：

```
>>> my_car[0]
'red'
>>> tuple(my_car)
('red', 3812.4)
```

元组拆包和用于函数参数拆包的`*`运算符也可以按预期工作：

```
>>> color, mileage = my_car
>>> print(color, mileage)
red 3812.4
>>> print(*my_car)
red 3812.4
```

你甚至可以免费为 `namedtuple` 对象获得漂亮的字符串表示形式，这节省了一些冗长的内容：

```
>>> my_car
Car(color='red', mileage=3812.4)
```

像元组一样，`namedtuple` 是不可变的。当你尝试覆盖其中一个字段，你会收到`AttributeError`异常：

```
>>> my_car.color = 'blue'
AttributeError: "can't set attribute"
```

在内部，`namedtuple` 对象通过常规Python类实现。在内存使用方面，它们也比常规类更好，并且具有与常规元组一样高的内存使用效率。

看待它们的一个好方法是认为 `namedtuple` 是在Python中手动定义不可变类的快捷方式。

继承具名元组

由于它们是基于常规Python类构建的，因此你甚至可以将方法添加到 `namedtuple` 对象。与其他任何类一样，你可以扩展 `namedtuple` 的类，并添加新方法和属性。这是一个例子：

```
Car = namedtuple('Car', 'color mileage')
```

```
class MyCarWithMethods(Car):
    def hexcolor(self):
        if self.color == 'red':
            return '#ff0000'
        else:
            return '#000000'
```

和预期的一样，现在我们可以创建MyCarWithMethods对象并调用它们的 hexcolor() 方法：

```
>>> c = MyCarWithMethods('red', 1234)
>>> c.hexcolor()
'#ff0000'
```

但是，这可能有些笨拙。如果你想要一个具有不变属性的类，你这样做可能值得，但是在*这里*也很容易搬起石头砸自己的脚。

考虑到 namedtuple 的内部实现方式，添加新的不可变字段非常棘手。创建 namedtuple 层次结构的最简单方法是使用基础tuple的_fields属性：

```
>>> Car = namedtuple('Car', 'color mileage')
>>> ElectricCar = namedtuple(
... 'ElectricCar', Car._fields + ('charge',))
```

这给出了期望的结果：

```
>>> ElectricCar('red', 1234, 45.0)
ElectricCar(color='red', mileage=1234, charge=45.0)
```

内建的帮助方法

除了 _fields 属性之外，每个 namedtuple 实例还提供了一些你可能会觉得有用的辅助方法。他们的名字全部以单个下划线字符开头，该字符通常表示方法或属性是“私有的”，而不是类或模块的稳定公共接口的一部分。

对于 namedtuple，下划线命名约定具有不同的含义。这些辅助方法和属性是 namedtuple 的公共接口。之所以这样命名是为了避免与用户定义的元组字段命名冲突。所以在需要时使用它们！

我想告诉你一些 namedtuple 的帮助方法可能会派上用场的场景。让我们从 _asdict() 方法开始，它将 namedtuple 的内容作为字典返回：

```
>>> my_car._asdict()
OrderedDict([('color', 'red'), ('mileage', 3812.4)])
```

这在生成JSON输出时避免在字段名称中引入错别字很有用，例如：

```
>>> json.dumps(my_car._asdict())
'{"color": "red", "mileage": 3812.4}'
```

另一个有用的帮助方法是 _replace()。它创建一个元组的浅拷贝副本，并允许你有选择地替换其中的一些字段：

```
>>> my_car._replace(color='blue')
Car(color='blue', mileage=3812.4)
```

最后， _make() 方法可用通过一个序列或可迭代对象创建新的 namedtuple 实例：

```
>>> Car._make(['red', 999])
Car(color='red', mileage=999)
```

什么时候使用具名元组

通过强制为数据使用更好的结构，具名元组是清理代码并使之更易读的一种简单方法。

例如，我发现它来与字典等数据类型相比，固定格式的 `namedtuple` 可以帮助我更清晰地表达自己的意图。通常，当我尝试这种重构时，我会为我面临的问题想到更好的解决方案。

在非结构化元组和字典上使用 `namedtuple` 也可以使我的同事的生活更轻松，因为他们使传递的数据在一定程度上可以自我说明。

另一方面，如果它们并不能帮助我编写“更简洁”和更易于维护的代码，那么我会尽量不使用 `namedtuple`。就像本书中展示的许多其他技术一样，过犹不及。

但是，如果谨慎使用它们，则命名元组无疑可以使你的Python代码更好，更具表现力。

重点

- `collection.namedtuple`是在Python中手动定义一个不可变的类的快捷方式，其内存使用效率很高。
- `namedtuple`可以通过强制执行易于理解的数据结构来保持代码整洁。
- `namedtuple`提供了一些有用的辅助方法，所有这些方法都以一个下划线开头，但它们属于公共接口，可以放心使用。

4.7 类变量和实例变量的陷阱

除了区分类方法和实例方法，Python的对象模型还可以区分类变量和实例变量。

这是一个重要的区别，但作为Python新手，也给我带来了麻烦。很久没花时间去从头开始理解这些概念。所以我早期的OOP实验充满了令人惊讶的行为和奇怪的bug。在本章中，我们将通过一些动手的例子解决关于这个主题的所有挥之不去的困惑。

就像我说的那样，Python对象上有两种数据属性：类变量和实例变量。

类变量在类定义内部声明（但在任何实例方法外部）。它们不受类的任何实例的束缚。类变量将其内容存储在类本身，从类创建的所有对象都共享一组相同的类变量。这意味着修改类变量将同时影响所有实例。

实例变量始终与特定的实例绑定。它们的内容不存储在类中，而是存储在由类创建的每个对象上。因此，一个实例变量的内容完全独立于另一个实例对象。因此，修改实例变量只会影响一个实例。

好吧，这是相当抽象的——是时候看一些代码了！让我们开始老掉牙的“狗的例子”……出于某种原因，面向对象的教程总是用汽车或宠物来说明他们的观点，很难打破那个传统。

快乐的狗需要什么？四条腿和一个名字：

```
class Dog:
    num_legs = 4 # <- Class variable

    def __init__(self, name):
        self.name = name # <- Instance variable
```

好吧，这就是我刚才描述的狗的面向对象的表示形式。创建新的Dog实例可以正常工作，每个实例都有一个名为name的实例变量：

```
>>> jack = Dog('Jack')
>>> jill = Dog('Jill')
>>> jack.name, jill.name
('Jack', 'Jill')
```

在类变量方面有更多的灵活性。你可以直接从每个Dog实例或从类本身访问num_legs类变量：

```
>>> jack.num_legs, jill.num_legs
(4, 4)
>>> Dog.num_legs
4
```

但是，如果你尝试通过该类访问实例变量，它会失败并抛出`AttributeError`。对每个实例对象都拥有独立的实例变量，实例变量在`__init__`初始化函数运行时创建——它们不存在于类本身。

这是类变量和实例变量之间的主要区别：

```
>>> Dog.name
AttributeError:
```

```
"type object 'Dog' has no attribute 'name'"
```

好吧，到目前为止一切都很好。假设狗狗杰克（Jack the Dog）有一天吃晚餐时，距离微波炉太近了，并且多了一对腿。你怎么用我们在小代码沙箱中的代码表示这种情景？

解决方案的第一个想法可能是简单地修改Dog类上的num_legs变量：

```
>>> Dog.num_legs = 6
```

但是请记住，我们不希望所有的狗都开始用六条腿四处乱窜。现在，由于修改了一个类变量，我们刚刚将小宇宙中的每个狗实例都变成了超级狗。这会影响所有狗，甚至包括以前创建的那些狗：

```
>>> jack.num_legs, jill.num_legs
(6, 6)
```

所以那没用。它不起作用的原因是修改了一个类名称空间上的类变量会影响该类的所有实例。让我们将更改回滚，并尝试仅给对杰克额外的一对腿：

```
>>> Dog.num_legs = 4
>>> jack.num_legs = 6
```

现在，这创造了什么怪物？让我们找出：

```
>>> jack.num_legs, jill.num_legs, Dog.num_legs
(6, 4, 4)
```

好吧，这看起来“不错”（除了我们刚才给可怜的杰克多了一些腿）。但是这种变化实际上是如何影响的我们的狗实例呢？

你会发现，这里的问题是，虽然我们得到了想要的结果（Jack额外的腿），但我们还是将num_legs实例变量引入了实例。现在，新的num_legs实例变量覆盖并隐藏了具有相同名称的类变量，当我们访问对象实例时：

```
>>> jack.num_legs, jack.__class__.num_legs
(6, 4)
```

你看，类变量没有被同步。发生这种情况是因为写入 jack.num_legs 时创建了一个与类变量同名实例变量。

这并不一定很糟糕，但重要的是要意识到幕后发生的事情。在我最终了解类级别的范围和实例级别的范围前，这是bug进入我的程序的绝佳途径。

实话实说，尝试通过对象修改类变量——然后意外地创建了一个同名实例变量，并且覆盖了类变量——有点像是Python面向对象的陷阱。

没有狗的例子

尽管本章的编写过程中没有狗受到伤害（直到有狗发芽长出额外的双腿时，这一切都很有趣），我想给你一个更实际的例子，说明那些你可以使用类变量做的有用的事情。一些关于类变量的更接近实战的东西。

就是这样。下面的CountedObject类跟踪在程序的整个生命周期中它被实例化了多少次（可能实际上是一个有趣的性能指标）：

```
class CountedObject:
    num_instances = 0

    def __init__(self):
        self.__class__.num_instances += 1
```

CountedObject保持一个num_instances类变量，该变量用作共享计数器。声明类后，它将初始化计数器为零。

每次你使用__init__构造函数创建此类的新实例时，共享计数器加1：

```
>>> CountedObject.num_instances
0
```

```
>>> CountedObject().num_instances
1
>>> CountedObject().num_instances
2
>>> CountedObject().num_instances
3
>>> CountedObject.num_instances
3
```

请注意，此代码需要确保它增加了类上的计数器变量。如果我按如下方式编写构造函数，则容易导致错误：

```
# WARNING: This implementation contains a bug
class BuggyCountedObject:
    num_instances = 0

    def __init__(self):
        self.num_instances += 1 # !!!
```

你看，这种（错误的）实现永远不会增加共享计数器变量：

```
>>> BuggyCountedObject.num_instances
0
>>> BuggyCountedObject().num_instances
1
>>> BuggyCountedObject().num_instances
1
>>> BuggyCountedObject().num_instances
1
>>> BuggyCountedObject().num_instances
0
```

我确定你可以看到我现在出了问题。这个有bug的实现在执行过程永远不会增加共享计数器，因为我犯了我在前面的“杰克狗”示例中解释的错误。这个实现无法正常进行，因为通过在构造函数中创建同名变量，我不小心“遮盖了”`num_instances`类变量。

它会正确计算计数器的新值（从0到1），但随后却将结果存储在了实例变量中——这意味着该类的其他实例甚至都不会看到更新后的计数器值。

你看，这是一个很容易犯的错误。在处理共享状态时小心并仔细检查范围会是一个好主意。自动化的测试和同行评审也会有很大的帮助。

不过，我希望你能看到为什么在实践中类变量可能是有用的工具以及如何使用它们——尽管他们有陷阱。祝你好运！

重点

- 类变量是类的所有实例共享的数据。它们属于一个类，而不是特定的实例，并且在该类的所有实例之间共享。
- 实例变量用于每个实例唯一的数据。它们属于单个实例对象，并且不在一个类的其他实例中共享。每个实例变量获取实例特定的值。
- 因为类变量可以被相同名称的实例变量“遮盖”，所以很容易（偶然）因为覆盖类变量而引入bug和奇怪的行为。

4.8 实例方法，类方法，静态方法揭秘

在本章中，你将了解类方法，静态方法，和Python中的实例方法。

如果你对它们之间的差异有了直观的了解，你将能够编写出更清晰传达意图的面向对象的Python程序，并且从长远来看也更容易维护。

首先，编写一个包含所有三种方法的简单示例（Python 3）类：

```
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Python 2 用户注意事项：@staticmethod 和 @classmethod 声明符从 Python 2.4 开始可用。你可以使用 MyClass(object) 语法，声明一个继承自 object 的新样式类，而不是使用 class MyClass 语句。

实例方法

MyClass 上的第一个方法称为 method，是常规实例方法。这将是你在大多数情况下使用的基本的简洁的方法类型。你可以看到该方法接受一个参数 self，该参数在调用该方法时指向 MyClass 的实例。当然，实例方法可以接受多个参数。

通过 self 参数，实例方法可以自由访问在同一对象上的属性和其他方法。这给了他们很多修改对象状态的能力。

它们不仅可以修改对象状态，实例方法还可以通过 self.__class__ 属性访问类本身。这意味着实例方法也可以修改类状态。

类方法

让我们将其与第二个方法 MyClass.classmethod 进行比较。我使用 @classmethod 装饰器将此方法标记为类方法。

类方法不接受 self 参数，而是接受 cls 参数，该参数在方法被调用时指向该类而不是其实例。

由于类方法只能访问此 cls 参数，因此无法修改对象实例状态，因为那需要 self 参数。但是，类方法仍然可以修改类状态，类状态会在所有实例中都被使用。

静态方法

使用 @staticmethod 装饰器将第三个方法 MyClass.staticmethod 标记为静态方法。尽管这种方法不带 self 或 cls 参数，但是可以使其接受任意数量的其他参数。

静态方法无法修改对象状态或类状态。静态方法在可以访问哪些数据方面受到限制——它们主要是为方法创造命名空间的一种方法。

让我们在实践中看看它们

我知道到目前为止，这种讨论还只是理论上的。我也相信对你而言，关于这些方法类型在实践中有何不同，有一个直观的理解是很重要的。这就是为什么我们要有一些具体的例子的原因。

让我们看一下这些方法在调用时的行为。我们先创建一个类的实例，然后调用三种不同的方法就可以了。

MyClass 的实现方式使得每种方法的实现返回一个元组，其中包含了我们可以用来跟踪正在发生的事情以及方法可以访问类或对象的哪些部分的信息。

当我们调用实例方法时，会发生以下情况：

```
>>> obj = MyClass()
>>> obj.method()
('instance method called', <MyClass instance at 0x11a2>)
```

这确认了在这种情况下，method 实例方法可以通过 self 参数访问对象实例（打印为）。

在方法被调用时，Python 将 self 参数替换为实例对象 obj。我们可以忽略 obj.method() 点调用语法提供的语法糖，并手动传递实例对象以获得相同的结果：

```
>>> MyClass.method(obj)
('instance method called', <MyClass instance at 0x11a2>)
```

顺便说一句，实例方法还可以通过 self.__class__ 属性访问类本身。这使得实例方法在访问限制方面具有强大的功能——它们可以自由地修改实例和类的状态。

接下来让我们尝试使用类方法：

```
>>> obj.classmethod()
('class method called', <class MyClass at 0x11a2>)
```

调用 classmethod() 告诉我们它无权访问 <MyClass instance> 对象，但仅限于代表类本身的 <class MyClass> 对象，（Python 中万物皆对象，甚至类也是对象）。

注意一下当我们调用 `MyClass.classmethod()` 时，Python如何将类作为第一个参数自动传递给函数。在Python中通过点语法调用一个方法时将触发此行为。实例方法上的 `self` 参数的工作方式相同。

请注意，将这些参数命名为 `self` 和 `cls` 只是一个约定。你可以轻松地将它们命名为`the_object`和`the_class`并获得相同的结果。唯一重要的是它们位于该特定方法的参数列表中的第一位。

现在该调用静态方法了：

```
>>> obj.staticmethod()
'static method called'
```

你是否看到我们如何在对象上调用 `staticmethod()` 并能成功吗？一些开发人员在了解到可以在对象实例上调用静态方法时感到惊讶。

在幕后，Python在使用点语法调用静态方法时通过不传递`self`或`cls`参数强制执行访问限制。

这证实了静态方法既不能访问实例状态也不能访问类状态。它们像常规函数一样工作，但是属于该类（以及每个实例的）名称空间。

现在，让我们来看看当我们在没有事先创建对象实例的情况下，尝试通过类本身调用这些方法会发生什么：

```
>>> MyClass.classmethod()
('class method called', <class MyClass at 0x11a2>)

>>> MyClass.staticmethod()
'static method called'

>>> MyClass.method()
TypeError: "unbound method method() must
be called with MyClass instance as first
argument (got nothing instead)"
```

我们能够调用 `classmethod()` 和 `staticmethod()`，但是尝试调用实例方法 `method()` 失败，并显示 `TypeError`。

这是预料之中的。这时我们没有创建对象实例并尝试直接在类上调用实例函数。这意味着 Python 无法填充 `self` 参数，因此调用失败，并出现 `TypeError` 异常。

这应该让这三种方法的区别稍微清楚一点。但请放心，我不会就这样停止。在在接下来的两个部分中，我将介绍两个关于什么时候使用这些特殊方法类型的更实际的示例。

我将以这个简单的披萨类为基础：

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

>>> Pizza(['cheese', 'tomatoes'])
Pizza(['cheese', 'tomatoes'])
```

拥有@classmethod的美味披萨工厂

如果你在现实世界中接触过披萨，就会知道有许多可口的变化：

```
Pizza(['mozzarella', 'tomatoes'])
Pizza(['mozzarella', 'tomatoes', 'ham', 'mushrooms'])
Pizza(['mozzarella'] * 4)
```

几个世纪前，意大利人弄清了他们的披萨分类方法，因此这些美味的比萨都有自己的名字。我们会充分利用这一点，并为我们的披萨类用户创建的披萨对象提供更好的接口。

一个不错的方法是将类方法用作我们可以制作的各种披萨的工厂函数：

```

class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

    @classmethod
    def margherita(cls):
        return cls(['mozzarella', 'tomatoes'])

    @classmethod
    def prosciutto(cls):
        return cls(['mozzarella', 'tomatoes', 'ham'])

```

注意我如何在 `margherita` 和 `prosciutto` 工厂方法使用 `cls` 参数，而不是直接调用 `Pizza` 的构造函数。

这是一个技巧，你可以用来遵循“不要重复自己”（DRY）原则。如果在某个时候我们决定重命名该类，我们不必记住要在所有工厂中更新构造函数名称。

现在，我们可以用这些工厂方法做什么？让我们尝试一下：

```

>>> Pizza.margherita()
Pizza(['mozzarella', 'tomatoes'])

>>> Pizza.prosciutto()
Pizza(['mozzarella', 'tomatoes', 'ham'])

```

你看，我们可以使用工厂函数来创建按我们想要的方式配置的新`Pizza`对象。他们都使用相同的 `__init__` 构造函数，只是提供一个快捷方式来记住所有各种成分。

看待类方法使用情况的另一种视角是认识到它们允许你为类定义替代的构造函数。

Python每个类只允许一个 `__init__` 方法。使用类方法可以根据需要添加尽可能多的替代构造函数。这可以使你的类的接口在一定程度上具有自记录功能并简化其使用。

什么时候使用静态方法

在这里想出一个很好的例子要困难一些，但是告诉你——我将继续把比萨类做得越来越薄。

这是我想出的：

```

import math

class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = ingredients

    def __repr__(self):
        return (f'Pizza({self.radius!r}, '
               f'{self.ingredients!r})')

    def area(self):
        return self.circle_area(self.radius)

    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi

```

现在我在这里修改了什么？首先，我修改了构造函数和 `__repr__` 以接受一个额外的 `radius` 参数。

我还添加了一个 `area()` 实例方法，该方法计算并返回披萨的面积。这也是一个 `@property` 很好的应用场景，但是，这只是一个玩具示例。

为了计算披萨的面积，通过使用众所周知的圆面积公式，我将其分解为一个单独的 `circle_area()` 静态方法而不是直接使用 `area()` 计算面积。试试吧！

```
>>> p = Pizza(4, ['mozzarella', 'tomatoes'])
>>> p
Pizza(4, {'self.ingredients'})
>>> p.area()
50.26548245743669
>>> Pizza.circle_area(4)
50.26548245743669
```

当然，这仍然是一个简单的例子，但这有助于解释静态方法提供的一些好处。

据我们了解，静态方法因为没有 `cls` 或 `self` 参数，所以无法访问类或实例状态。这是一个很大的限制——但这是一个很好的信号，它标明了一个与周围的其他事物无关的特定方法。

很明显，在上面的示例中，`circle_area()` 无法以任何方式修改类或类实例。（当然，你可以随时用一个全局变量来解决这个问题，但这不是重点。）

为什么这样做有用？

将方法标记为静态方法不仅暗示方法不会修改类或实例状态。此限制也在 Python 运行时强制执行。

这样的技巧可以使你清楚地交流类架构的不同部分，因此新的开发工作自然被引导在这些边界内发生。当然，克服这些限制很容易。但实际上，它们通常会对避免不符合原始设计的意外修改有所帮助。

换句话说，使用静态方法和类方法是可以在充分执行开发人员意图的同时传达开发人员的意图以避免大多数因疏忽导致的 bug 和错误，这些 bug 和错误会破坏设计。

在有意义的时候，写一些这样的方法可以提高可维护性并减少维护成本，可以减少其他开发人员错误地使用了你的类的可能。

在编写测试代码时，静态方法也有好处。由于 `circle_area()` 方法完全独立于类的其余部分，它更容易测试。

在单元测试中测试方法之前，我们不必担心构造一个完整的类实例。我们可以像测试一个常规函数一样进行测试。再次，这使未来的维护更容易，并提供了面向对象和面向过程编程风格的一个连接。

重点

- 实例方法需要一个类实例，并且可以通过 `self` 来访问该实例。
- 类方法不需要类实例。他们无法访问实例，但是他们可以通过 `cls` 访问类本身。
- 静态方法无权访问 `cls` 或 `self`。他们像常规函数，但属于该类的名称空间。
- 静态方法和类方法进行通信，并且（在一定程度上）加强开发人员对类设计的意图。这可以带来明确的维护优势。

source/python_tricks/5_通用数据结构.md

第五章 Python中常见数据结构

每个Python开发人员都应该学习和练习的是什么？

数据结构。

它们是构建程序的基石。每个数据结构提供一种特定的组织数据的方式，以便根据使用场景有效地访问数据。

我相信，作为开发，无论他们的技术水平或经验如何，回到基础总是会有所回报。

现在，我不主张你应该仅仅专注于扩展数据结构知识——这是一种“失败模式”，停留在理论上，从不产出任何东西.....

但是我发现，花一些时间整理数据结构和算法知识总能带来回报。

无论是几天时间的“冲刺”，还是一个在不同地方都投入一点时间的项目，都没有什么区别。无论哪种方式，这都需要花费大量时间。

好了，Python中的数据结构？我们有列表，字典，集合...嗯，栈？我们有栈吗？

你会发现，麻烦在于Python在标准库中附带了大量数据结构。但是，有时命名有点“离谱”。

经常不清楚像栈这种常见的“抽象数据类型”怎么对应到Python中的一个特定实现。Java之类的其他语言则坚持一种更加符合“计算机科学”的明确命名方案：列表不仅仅叫 `list` ——要么是 `LinkedList` 要么是 `ArrayList`。

这让我们可以更容易识别这些类型的预期行为和复杂度。Python坚持更简单和更“人性化”的命名方案，我喜欢它。在某种程度上，这使使用Python进行编程变得非常有趣。

但是缺点是，即使是经验丰富的Python开发人员，也可能不清楚内置列表底层实现到底是链表还是动态数组。而有一天，这些知识的缺乏将使他们感到沮丧，或者在求职面试中被拒绝。

在本书的这一部分中，你将看到基本数据结构和抽象数据类型（ADT）在Python及其标准库的实现。

我的目标是阐明最常见的抽象数据类型在Python的命名方案并为每种数据类型提供一个简短的说明。这些信息还将帮助你在Python面试编码中大放异彩。

如果你正在寻找一本好书来补充你的通用数据结构知识，我强烈推荐史蒂文·S·斯基纳（Steven S. Skiena）的《算法》设计手册

它在教导你基础数据结构（包括更高级的数据结构），和向你展示如何将其实际应用到各种算法之间取得了很好地平衡。史蒂夫的书在编写这些章节时提供了很大帮助。

5.1 字典，映射和哈希表

在Python中，字典（简称为“dict”）是一个核心数据结构。字典可以存储任意数量的对象，每个对象都由一个唯一的键表示。

字典通常也称为映射，哈希表。它们可以对给定键对应的任何对象进行高效的查找，插入和删除操作。

在实践中，这有什么意义呢？事实证明字典类似于真实世界中的电话簿：

电话簿使你可以快速检索与给定的键（一个人的名字）相关的信息（电话号码）。因此，不必从前到后查阅读整个电话簿来查找某个人的电话号码，你可以直接跳到一个名字，并查找相关信息。

当讨论到怎么组织信息以保证高效查找时，这种类比在某种程度上就失效了。但是基本的性能特征仍然存在：字典使你能够快速找到与给定的键关联的信息。

总而言之，字典是计算机科学中最重要最常用的数据结构之一。

那么，Python如何处理字典呢？

让我们浏览一下Python中的字典实现。

字典

由于其重要性，Python有直接内置于核心的很健壮的字典实现。

Python还提供了一些关于字典的“语法糖”。例如，大括号字典语法和字典解析可以使你方便的定义新的字典对象：

```
phonebook = {
    'bob': 7387,
    'alice': 3719,
    'jack': 7052,
}

squares = {x: x * x for x in range(6)}

>>> phonebook['alice']
3719

>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

哪些对象可以用作有效键是有一些限制的。

Python的字典由可哈希的键索引：可哈希对象的哈希值在存在期间不会发生变化（请参见 `__hash__`），并且可以将其与其他对象进行比较（请参见 `__eq__`）。此外，可哈希对象相等必须具有相同的哈希值。

诸如字符串和数字之类的不可变类型是可哈希的。如果元组本身仅包含可哈希类型，你还可以将其用作字典键。

在大多数情况下，Python的内置字典实现已经做了你想做的一切。字典经过了高度优化，并且是Python许多部分的基础，例如类属性和栈框架中的变量都存储在字典中。

Python字典基于充分测试和精心调整的哈希表实现，可以提供你期望的性能：

$O(1)$ 时间复杂度实现查找、插入、更新和删除操作。

没有理由不使用Python附带的标准dict实现。但是，存在专门的第三方字典实现，例如，基于跳表或基于B树的字典。

除了“普通”字典对象外，Python的标准库还包括一些特殊的字典实现。这些特殊的字典全部基于内置的字典类实现（具有同样的性能），但增加了一些便利功能。

让我们来看看它们。

collections.OrderedDict 保存key的插入顺序

Python有一个专门的dict子类，该子类保存了添加到它的键的插入顺序：`collections.OrderedDict`。

虽然在CPython 3.6及更高版本中标准dict实现保存了键的插入顺序，但这只是CPython实现的一个副作用，并未在语言规范中定义。因此，如果键的顺序对于你的算法正常运行很重要，最好通过显式使用 `OrderDict` 类清楚地表明这一点。

顺便说一句，`OrderedDict`不是核心语言的内置部分，并且必须从标准的 `collections` 模块中导入。

```
>>> import collections
>>> d = collections.OrderedDict(one=1, two=2, three=3)
>>> d
OrderedDict([('one', 1), ('two', 2), ('three', 3)])
>>> d['four'] = 4
>>> d
OrderedDict([('one', 1), ('two', 2),
('three', 3), ('four', 4)])
>>> d.keys()
odict_keys(['one', 'two', 'three', 'four'])
```

collections.defaultdict 键不存在时，返回默认值

`defaultdict` 是字典的另一个子类，其构造函数中可以接受一个可调用对象，如果找不到请求的key，可以用这个可调用对象产生默认值。

与使用 `get()` 方法或捕获常规字典中的 `KeyError` 异常相比，这样可以节省一些工作量，并可以让程序员的意图更加清晰。

```
>>> from collections import defaultdict
>>> dd = defaultdict(list)
```

```
# Accessing a missing key creates it and
# initializes it using the default factory,
# i.e. list() in this example:
>>> dd['dogs'].append('Rufus')
>>> dd['dogs'].append('Kathrin')
>>> dd['dogs'].append('Mr Sniffles')
>>> dd['dogs']
['Rufus', 'Kathrin', 'Mr Sniffles']
```

collections.ChainMap 组合多个字典

`collections.ChainMap` 将多个字典组合到一个字典中。查找操作会在基础字典中一个个进行搜索，直到找到键。插入，更新和删除操作仅影响添加到链中的第一个字典。

```
>>> from collections import ChainMap
>>> dict1 = {'one': 1, 'two': 2}
>>> dict2 = {'three': 3, 'four': 4}
>>> chain = ChainMap(dict1, dict2)
>>> chain
ChainMap({'one': 1, 'two': 2}, {'three': 3, 'four': 4})
# ChainMap searches each collection in the chain
# from left to right until it finds the key (or fails):
>>> chain['three']
3
>>> chain['one']
1
>>> chain['missing']
KeyError: 'missing'
```

types.MappingProxyType 实现只读字典的封装

`MappingProxyType` 是标准字典的封装，提供字典数据的只读视图。这个类在Python 3.3中添加，可用于创建字典的不可变代理版本。

例如，如果你希望从类或模块中返回包含内部状态的字典，而又不想开放写权限，那么这可能会有所帮助。使用 `MappingProxyType` 可让你在原地实现这些限制，而无需创建该字典的完整副本。

```
>>> from types import MappingProxyType
>>> writable = {'one': 1, 'two': 2}
>>> read_only = MappingProxyType(writable)
# The proxy is read-only:
>>> read_only['one']
1
>>> read_only['one'] = 23
TypeError:
"'mappingproxy' object does not support item assignment"
# Updates to the original are reflected in the proxy:
>>> writable['one'] = 42
>>> read_only
mappingproxy({'one': 42, 'two': 2})
```

字典总结

本章列出的所有Python字典实现都是Python标准库中内置的有效实现。

如果你正在寻找关于要在你的程序中使用哪种映射的建议，我将向你推荐内置的`dict`数据类型。这是一种通用且经过优化的哈希表实现，且已经直接内置到了核心语言中。

只有你有 `dict` 所提供的特性以外的其它特殊要求时，我才会建议你使用这里列出的其他数据类型里面的一种。

是的，我仍然相信所有这些实现都是有效的，但是如果大多数时候都使用Python标准的字典实现，那么你的代码将更加清晰，并且更容易被其他开发者维护。

重点

- 字典是Python中的核心数据结构。
- 大多数时候，内置的 `dict` 类型是“足够好”的。

- 在Python标准库中有像只读或有序字典之类的特殊实现。

5.2 数组类型

数组是大多数编程语言中的基本数据结构，并且在不同的算法中具有广泛的应用。

在本章中，我们将介绍Python标准库中的数组实现。

你会看到每种方法的优缺点，因此你可以确定哪种实现适合你的场景。但是在我们开始之前，让我们先介绍一些基础知识。

数组如何工作，它们的作用是什么？

数组由固定大小的数据记录组成，它允许根据索引对每个元素进行有效定位。

由于数组将信息存储在相邻的内存块中，它们被认为是连续的数据结构（而不是链接的数据结构，例如链表。）

数组数据结构的一个现实世界中的类比是一个停车场：

你可以从整体上看待停车场并将其视为一个物体，但在内有由唯一编号索引的停车位。停车位是车辆的容器——每个停车位可以为空或有汽车，摩托车或其他车辆停放在上面。

但并非所有停车场都一样：

一些停车场可能只限于一种车辆。例如，汽车停车场不会允许将自行车停在上面。“受限”停车场对应于“类型数组”数据结构，该数据结构仅允许存储具有相同数据类型的元素。

从性能角度来看，通过索引查找其中的元素非常快。正确的数组实现保证在这种场景下拥有常数时间复杂度 $O(1)$ 。

Python标准库中包括几个类似数组的数据结构，每个结构都具有略有不同。让我们来看看：

list 可变的动态数组

列表是Python核心语言的一部分。尽管这样命名，在底层Python的列表是通过动态数组实现的。这意味着一个列表允许添加或删除元素，并且通过分配或释放内存，该列表会自动调整包含这些元素的后备存储。

Python列表可以包含包括函数在内的任意元素——Python中一切皆对象。因此，你可以混合不同类型的数据，并将它们全部存储在一个列表中。

这是一个强大的功能，但缺点是支持同时使用多种数据类型意味着数据封装通常不够紧密。整个结构会占用更多的空间。

```
>>> arr = ['one', 'two', 'three']
>>> arr[0]
'one'

# Lists have a nice repr:
>>> arr
['one', 'two', 'three']

# Lists are mutable:
>>> arr[1] = 'hello'
>>> arr
['one', 'hello', 'three']

>>> del arr[1]
>>> arr
['one', 'three']

# Lists can hold arbitrary data types:
>>> arr.append(23)
>>> arr
['one', 'three', 23]
```

元组，不可变容器

就像列表一样，元组也是Python核心语言的一部分。与列表不同，Python的元组对象是不可变的。这意味着元素不能动态添加或删除元素——元组中的所有元素都必须在创建时定义。

就像列表一样，元组可以容纳任意数据类型的元素。有这种灵活性很强大，但同样，这也意味着与类型数组相比数据封装并不紧密。

```
>>> arr = 'one', 'two', 'three'
>>> arr[0]
'one'

# Tuples have a nice repr:
>>> arr
('one', 'two', 'three')

# Tuples are immutable:
>>> arr[1] = 'hello'
TypeError:
"'tuple' object does not support item assignment"

>>> del arr[1]
TypeError:
"'tuple' object doesn't support item deletion"
# Tuples can hold arbitrary data types:
# (Adding elements creates a copy of the tuple)

>>> arr + (23,)
('one', 'two', 'three', 23)
```

array.array 基础的类型数组

Python的数组模块提供了对于像字节，32位整数，浮点数这些基本的C风格数据类型的存储方式，这种方式具有很高的空间利用率。

用array.array类创建的数组是可变的并且其行为和 list 类似，但有一个重要区别——它们是类型数组，仅包含一种数据类型。

因为这个约束，具有许多元素的 array.array 对象比列表和元组更节省空间。存储在其中的元素紧密封装，如果你需要存放许多相同类型的元素，这可能会很有用。

此外，数组支持许多与常规列表相同的方法，因此你可能可以将它们用作列表的“替代品”，而无需对代码进行其他更改。

```
>>> import array
>>> arr = array.array('f', (1.0, 1.5, 2.0, 2.5))
>>> arr[1]
1.5

# Arrays have a nice repr:
>>> arr
array('f', [1.0, 1.5, 2.0, 2.5])

# Arrays are mutable:
>>> arr[1] = 23.0
>>> arr
array('f', [1.0, 23.0, 2.0, 2.5])

>>> del arr[1]
>>> arr
array('f', [1.0, 2.0, 2.5])

>>> arr.append(42.0)
>>> arr
array('f', [1.0, 2.0, 2.5, 42.0])

# Arrays are "typed":
>>> arr[1] = 'hello'
TypeError: "must be real number, not str"
```

str unicode字符的不可变数组

Python 3.x使用 str 对象将文本数据存储为不可变的Unicode字符序列。实际上，这意味着 str 是一个不可变的字符数组。奇怪的是，它也是一种递归数据结构——字符串中的每个字符都是长度为1的一个 str。

字符串对象空间利用率很高，因为它们紧密包装并且他们专注于一种数据类型。如果你要存储Unicode文本，你应该使用它们。由于字符串在Python中是不可变的，因此修改字符串需要创建修改后的副本。最接近的等价于“可变字符串”的实现是将单个字符存储在列表中。

```

>>> arr = 'abcd'
>>> arr[1]
'b'

>>> arr
'abcd'
# Strings are immutable:

>>> arr[1] = 'e'
TypeError:
"'str' object does not support item assignment"

>>> del arr[1]
TypeError:
"'str' object doesn't support item deletion"

# Strings can be unpacked into a list to
# get a mutable representation:
>>> list('abcd')
['a', 'b', 'c', 'd']
>>> ''.join(list('abcd'))
'abcd'

# Strings are recursive data structures:
>>> type('abc')
<class 'str'>
>>> type('abc'[0])
<class 'str'>

```

字节串 单个字节的不可变数组

字节对象是单个字节的不可变序列（范围为 $0 \leq x \leq 255$ ）。从概念上讲，它们类似于 str 对象，你也可以将它们视为字节的不可变数组。

像字符串一样，字节具有自己的创建语法，而且它们节省空间。字节对象是不可变的，但与字符串不同，可以将字节串拆包为一种称为“可变字节数组”的专用数据类型。在下一节你会接触到更多相关信息。

```

>>> arr = bytes((0, 1, 2, 3))
>>> arr[1]
1

# Bytes literals have their own syntax:
>>> arr
b'\x00\x01\x02\x03'
>>> arr = b'\x00\x01\x02\x03'

# Only valid "bytes" are allowed:
>>> bytes((0, 300))
ValueError: "bytes must be in range(0, 256)"
# Bytes are immutable:
>>> arr[1] = 23
TypeError:
"'bytes' object does not support item assignment"

>>> del arr[1]
TypeError:
"'bytes' object doesn't support item deletion"

```

字节数组——可变的字节数组

bytearray 类型是一个范围为 $0 \leq x \leq 255$ 的可变整数序列。它们与 bytes 的主要区别在于字节数组可以自由修改，你可以覆盖元素，删除现有元素或添加新元素。 bytearray 对象将相应地进行扩缩容。

字节数组可以转换回不可变字节对象，但是这涉及到全量复制数据，这是一个很慢的操作，时间复杂度是 $O(n)$.

```

>>> arr = bytearray((0, 1, 2, 3))
>>> arr[1]
1

```

```
# The bytearray repr:
>>> arr
bytearray(b'\x00\x01\x02\x03')
# Bytearrays are mutable:
>>> arr[1] = 23
>>> arr
bytearray(b'\x00\x17\x02\x03')
>>> arr[1]
23
# Bytearrays can grow and shrink in size:
>>> del arr[1]
>>> arr
bytearray(b'\x00\x02\x03')
>>> arr.append(42)
>>> arr
bytearray(b'\x00\x02\x03*')
# Bytearrays can only hold "bytes"
# (integers in the range 0 <= x <= 255)
>>> arr[1] = 'hello'
TypeError: "an integer is required"
>>> arr[1] = 300
ValueError: "byte must be in range(0, 256)"
# Bytearrays can be converted back into bytes objects:
# (This will copy the data)
>>> bytes(arr)
b'\x00\x02\x03*'
```

重点

在Python中需要实现数组时，你可以选择许多内置数据结构。在本章中，我们已经聚焦于标准库中的核心语言功能和数据结构。

如果你愿意跳出Python标准库，那么可以使用像NumPy之类第三方库，它为科学计算和数据科学提供了广泛的快速数组实现。

如果限制到Python包含的数组类型数据结构，以下是我们的选择：

你需要存储任意对象，可能混合使用数据类型？根据你想要可变还是不可变的数据结构，确定使用列表还是元组。

你有数字类型的数据（整数或浮点数）并且紧密包装和性能很重要？试试 `array.array` 看看它是否可以满足你的所有需求。另外，考虑一下标准库之外的库，试试 NumPy 或 Pandas 之类的包。

你有以Unicode字符表示的文本数据吗？使用Python内置的 `str`。如果你需要“可变字符串”，请使用一个列表保存字符串。

你要存储一个连续的字节块？使用不可变的字节类型，如果需要可变的数据结构，则使用字节数组。

在大多数情况下，我喜欢从一个简单的列表开始。稍后，我会研究性能或存储空间是否是一个问题。大多数情况下，使用像 `list` 这样的通用数组数据结构可以给你最快的开发速度和最大的编程便利性。

我发现与试图从一开始就压榨最后一点性能相比，这通常更重要。

5.3 记录，结构体和数据传输对象

与数组相比，记录数据结构提供了固定数量的字段，其中每个字段可以有一个名称，并且可以有不同的类型。

在本章中，你将了解如何仅使用内置数据类型和标准库中的类实现记录，结构体和“普通的旧数据对象”。

顺便说一句，我在这里使用记录的宽松定义。例如，我还讨论诸如Python的内置元组之类的类型，从严格意义上来说，因为它们不提供命名字段，可能会或也可能不会将其视为记录。

Python提供了几种可用于实现记录的数据类型，结构体和数据传输对象。在本章中，你将快速获得了解每个实现及其独特的特性。在最后，你会找到总结和决策指南，这将对你有所帮助。

好了，我们开始！

dict ——简单的数据对象

Python字典存储任意数量的对象，每个对象用唯一的键标识。字典通常也称为映射或哈希表，并允许通过给定的键实现高效的查找，插入和删除操作。

在Python中使用字典作为记录数据类型或数据对象是可能的。字典很容易在Python中创建，因为它们具有自己的语法糖。字典语法简洁明了，写起来非常方便。

使用字典创建的数据对象是可变的，几乎没有防止字段名拼写错误的机制，任何时候都可以添加和删除字段。这两个属性都可以引入令人惊讶的bug，并且始终需要在便利性和错误恢复能力之间进行权衡。

```

car1 = {
    'color': 'red',
    'mileage': 3812.4,
    'automatic': True,
}
car2 = {
    'color': 'blue',
    'mileage': 40231,
    'automatic': False,
}
# Dicts have a nice repr:
>>> car2
{'color': 'blue', 'automatic': False, 'mileage': 40231}
# Get mileage:
>>> car2['mileage']
40231
# Dicts are mutable:
>>> car2['mileage'] = 12
>>> car2['windshield'] = 'broken'
>>> car2
{'windshield': 'broken', 'color': 'blue',
'automatic': False, 'mileage': 12}
# No protection against wrong field names,
# or missing/extra fields:
car3 = {
    'colr': 'green',
    'automatic': False,
    'windshield': 'broken',
}

```

元组，对象的不可变组合

Python的元组是用于对任意对象进行组合的简单数据结构。元组是不可变的，一旦被创建，就无法对其进行修改。

在性能方面，元组占用的内存比列表少一点点，并且构建起来也更快。

正如你在下面的反汇编字节码中所看到的，构造一个元组常量时只需要一个 LOAD_CONST 操作码，构建具有相同内容的列表对象需要执行更多操作：

```

>>> import dis
>>> dis.dis(compile("(23, 'a', 'b', 'c')", '', 'eval'))
0 LOAD_CONST 4 ((23, 'a', 'b', 'c'))
3 RETURN_VALUE
>>> dis.dis(compile("[23, 'a', 'b', 'c']", '', 'eval'))
0 LOAD_CONST 0 (23)
3 LOAD_CONST 1 ('a')
6 LOAD_CONST 2 ('b')
9 LOAD_CONST 3 ('c')
12 BUILD_LIST 4
15 RETURN_VALUE

```

但是，你不应过分强调这些差异。在实践中，性能差异通常可以忽略不计，试图通过从列表切换到元组以压榨额外的性能很可能是一个错误的方法。

普通元组的潜在缺点是你存储在其中的数据只能通过整数索引的方式进行访问。你不能为存储在元组中的各个属性命名。这个可以影响代码的可读性。

另外，元组始终是一种临时结构：很难确保两个元组具有相同的字段数，并且相同的属性存储在它们中。

这样可以很容易因为疏忽导致bug，例如混淆字段顺序。因此，我建议你在元组中存储的字段数越少越好。

```
# Fields: color, mileage, automatic
>>> car1 = ('red', 3812.4, True)
>>> car2 = ('blue', 40231.0, False)
# Tuple instances have a nice repr:
>>> car1
('red', 3812.4, True)
>>> car2
('blue', 40231.0, False)
# Get mileage:
>>> car2[1]
40231.0
# Tuples are immutable:
>>> car2[1] = 12
TypeError:
"'tuple' object does not support item assignment"
# No protection against missing/extraneous fields
# or a wrong order:
>>> car3 = (3431.5, 'green', True, 'silver')
```

写一个自己的类，做更多的工作，进行更好地控制

通过类，你可以为数据对象定义可重用的“蓝图”确保每个对象都提供相同的字段集。

使用常规的Python类作为记录数据类型是可行的，但还是需要手动工作才能获得其他实现的便利功能。例如，向 `__init__` 构造函数添加新字段冗长且需要时间。

另外，自定义类实例化对象的默认字符串表示形式并不是很有帮助。要解决此问题，你可能必须添加你自己的 `__repr__` 方法，这通常也很冗长并且每次添加新字段时都必须更新。

存储在类上的字段是可变的，可以自由地添加新字段，你可能喜欢也可能不喜欢。有可能使用 `@property` 装饰器提供更多访问控制并创建只读字段，但这又需要编写更多的胶水代码。

每当你想添加自定义类时，使用方法将业务逻辑和行为添加到记录对象中是一个不错的选择。但是，这意味着这些对象不再是普通的数据对象。

```
class Car:
    def __init__(self, color, mileage, automatic):
        self.color = color
        self.mileage = mileage
        self.automatic = automatic

>>> car1 = Car('red', 3812.4, True)
>>> car2 = Car('blue', 40231.0, False)
> # Get the mileage:
>>> car2.mileage
40231.0
# Classes are mutable:
>>> car2.mileage = 12
>>> car2.windshield = 'broken'
# String representation is not very useful
# (must add a manually written __repr__ method):
>>> car1
<Car object at 0x1081e69e8>
```

`collections.namedtuple` 方便的数据对象

Python 2.6+ 中新增的 `namedtuple` 类提供了对内置元组数据类型的扩展。与定义自定义类相似，使用 `namedtuple` 可以为你的数据记录定义可重用的“蓝图”，以确保使用正确的字段名称。

就像常规元组一样，命名元组是不可变的。这意味着在 `namedtuple` 实例创建之后不能添加新字段或修改现有字段。

除此之外，命名元组是有名字的元组。每个存储在其中的对象可以通过唯一标识符进行访问。这使你不必记住整数索引，也不必使用一些变通方法，将整数常量定义为助记符索引。

在内部，`namedtuple` 对象通过常规Python类实现。在内存使用方面，它们比常规类“更好”一点，并且拥有和常规元组一样高效的内存使用率：

```
>>> from collections import namedtuple
>>> from sys import getsizeof
>>> p1 = namedtuple('Point', 'x y z')(1, 2, 3)
>>> p2 = (1, 2, 3)
>>> getsizeof(p1)
72
>>> getsizeof(p2)
72
```

通过强制实现一个更好的数据结构，`namedtuple` 是保持代码整洁并提高其可读性的简单方法。

例如，我发现与字典等数据类型相比，固定格式的`namedtuple` 可以帮助我更清晰地表达自己的意图。通常，当我尝试这种重构时，我会为我面临的问题想到更好的解决方案。

在非结构化元组和字典上使用`namedtuple` 也可以使我同事的生活更轻松，因为他们使传递的数据在一定程度上可以自我说明。

```
>>> from collections import namedtuple
>>> Car = namedtuple('Car', 'color mileage automatic')
>>> car1 = Car('red', 3812.4, True)
# Instances have a nice repr:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)
# Accessing fields:
>>> car1.mileage
3812.4
# Fields are immutable:
>>> car1.mileage = 12
AttributeError: "can't set attribute"
>>> car1.windshield = 'broken'
AttributeError:
"'Car' object has no attribute 'windshield'"
```

typing.NamedTuple 改进的具名元组

Python 3.6中添加的这个类是`collections` 模块中的`namedtuple` 类的新兄弟。它与`namedtuple` 非常相似，主要区别是定义新记录类型时的更新语法，此外还增加了类型提示。

请注意，如果没有像`mypy` 这样单独的类型检查工具，类型注释不会强制执行。但是即使没有工具的支持，他们可以为其他程序员提供有用的提示（如果类型提示已过时，则会让他们感到非常困惑。）

```
>>> from typing import NamedTuple
class Car(NamedTuple):
    color: str
    mileage: float
    automatic: bool
>>> car1 = Car('red', 3812.4, True)
# Instances have a nice repr:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)
# Accessing fields:
>>> car1.mileage
3812.4
> # Fields are immutable:
>>> car1.mileage = 12
AttributeError: "can't set attribute"
>>> car1.windshield = 'broken'
AttributeError:
"'Car' object has no attribute 'windshield'"
# Type annotations are not enforced without
# a separate type checking tool like mypy:
>>> Car('red', 'NOT_A_FLOAT', 99)
Car(color='red', mileage='NOT_A_FLOAT', automatic=99)
```

struct.Struct 序列化的c语言结构体

`struct.Struct` 类在Python值和序列化为Python字节对象的C结构体之间转换。例如，它可以用于处理存储在文件或来自网络的二进制数据。

`struct` 是使用类似于格式字符串的迷你语言定义的，它允许你定义各种C数据类型（例如 `char`，`int` 和 `long`，以及它们的无符号变体）。

序列化结构很少用于表示完全可以在Python代码中处理的数据。它们主要是被用作数据交换格式，而不是仅仅在Python代码将数据保存在内存中的一种方式。

在某些情况下，将原始数据打包到 `struct` 中可能会比保存在其他数据类型中使用更少的内存。但是，在大多数情况下将是一个非常高级的（可能是不必要的）优化。

types.SimpleNamespace 精彩的属性访问

这是在Python中实现数据对象的另一种“神秘”选择：`types.SimpleNamespace`。该类是在Python 3.3中添加的，它提供对其名称空间的属性访问。

这意味着 SimpleNamespace 实例将其所有键暴露为类属性。这意味着你可以使用对象的“点操作”访问属性 (`obj.key`)，而不必使用常规命令所使用的方括号语法 (`obj['key']`)。默认情况下，所有实例还包括一个有意义的 `__repr__` 实现。

顾名思义，SimpleNamespace 很简单！基本上它是一个允许访问属性并进行打印的字典。可以随意添加，修改和删除属性。

```
>>> from types import SimpleNamespace
>>> car1 = SimpleNamespace(color='red',
... mileage=3812.4,
... automatic=True)
# The default repr:
>>> car1
namespace(automatic=True, color='red', mileage=3812.4)
# Instances support attribute access and are mutable:
>>> car1.mileage = 12
>>> car1.windshield = 'broken'
>>> del car1.automatic
>>> car1
namespace(color='red', mileage=12, windshield='broken')
```

重点

现在，你应该使用哪种数据类型？实现记录或数据对象有很多不同的方案。通常，你的决定取决于你的场景：

你只有几个 (2-3) 字段：如果字段顺序易于记忆或字段名称是多余的，使用普通的元组对象可能就可以了。例如，3D空间中的 (x, y, z) 点。

你需要不可变字段：在这种情况下，元组、`collections.namedtuple`、`typing.NamedTuple` 都是实现这种类型的数据对象的良好选择。

你需要锁定字段名称以避免输入错误: `collections.namedtuple` 和 `typing.NamedTuple` 是很好的选择。

你想使事情保持简单：考慮到和JSON类似的便捷的语法，普通的字典对象是一个不错的选择。

你需要完全控制自己的数据结构：是时候编写一个使用 `@property` 装饰器以及其 `setter` 和 `getter` 装饰器的自定义类了。

你需要向对象添加方法：应该写一个自定义类，要么从头开始实现，要么通过扩展 `collections.namedtuple` 或 `typing.NamedTuple` 实现。

你需要紧密打包数据以将其序列化到磁盘或通过网络进行发送：是时候阅读 `struct-Struct` 了，因为这是一个很好的用例。

如果你正在寻找安全的默认选择，我一般建议在Python 2.x及其版本中使用 `collections.namedtuple`，在Python 3中使用 `typing.NamedTuple`。

5.4 集合和多重集合

在本章中，你将了解如何在Python中使用标准库中的内置数据类型实现可变和不可变集合。首先，让我们快速回顾一下集合是什么：

集合是不允许重复元素的无序集合。通常，集合用于快速测试元素在集合中是否存在，从集合中插入或删除新值，以及计算两个集合的并集或交集。

在正确的集合实现中，测试元素是否存在时间复杂度是O(1)。并集，交集，差和子集运算平均时间复杂度是O(n)。Python的标准库中的集合实现符合上面所说的性能特征。

就像字典一样，集合在Python中得到了特殊对待，并且有一些语法糖来快速创建集合。例如，花括号语法和集合解析允许你方便地定义新的集合实例：

```
vowels = {'a', 'e', 'i', 'o', 'u'}
squares = {x * x for x in range(10)}
```

但请注意：创建一个空集合时，你需要调用 `set()` 函数。使用空的花括号 {} 语义并不清晰，它会创建一个空的字典。

Python及其标准库提供了几种集合实现。让我们来看看它们。

set 集合

这是Python内置的set实现。set类型是可变的，允许动态插入和删除元素。

Python的集合和 dict 具有相同的性能。任何可哈希的对象都可以存储在集合中。

```
>>> vowels = {'a', 'e', 'i', 'o', 'u'}
>>> 'e' in vowels
True
>>> letters = set('alice')
>>> letters.intersection(vowels)
{'a', 'e', 'i'}
>>> vowels.add('x')
>>> vowels
{'i', 'a', 'u', 'o', 'x', 'e'}
>>> len(vowels)
6
```

frozenset 不可变集合

`frozenset` 类实现了 `set` 的不可变版本，不可变集合创建后无法更改。不可变集合是静态的，仅允许对其元素执行查询操作（无插入或删除）。由于不可变集合是静态且可哈希的，因此它们可以用作字典的键或当做另一个集合的元素，这在可变集合中是不可能的。

```
>>> vowels = frozenset({'a', 'e', 'i', 'o', 'u'})
>>> vowels.add('p')
AttributeError:
"'frozenset' object has no attribute 'add'"
# Frozensets are hashable and can
# be used as dictionary keys:
>>> d = { frozenset({1, 2, 3}): 'hello' }
>>> d[frozenset({1, 2, 3})]
'hello'
```

collections.Counter 多重集合

Python标准库中的 `collections.Counter` 类实现了一种多重集合，它允许集合中的元素出现不止一次。

如果你不仅仅需要知道元素是否在集合中存在，还需要知道元素出现的次数，这个类型就派上了用场：

```
>>> from collections import Counter
>>> inventory = Counter()
>>> loot = {'sword': 1, 'bread': 3}
>>> inventory.update(loot)
>>> inventory
```

```

Counter({'bread': 3, 'sword': 1})
>>> more_loot = {'sword': 1, 'apple': 1}
>>> inventory.update(more_loot)
>>> inventory
Counter({'bread': 3, 'sword': 2, 'apple': 1})

```

这是使用 Counter 类的一个警告：

在计算 Counter 对象中元素的数量时你需要格外小心。调用 `len()` 会返回多重集中唯一元素的数量，使用 `sum()` 函数可以获取元素总数：

```

>>> len(inventory)
3 # Unique elements
>>> sum(inventory.values())
6 # Total no. of elements

```

重点

- 集合是Python及其标准库中包含的另一种常用的数据结构。
- 如果需要可变集合，请使用内置集合类型。
- `frozenset` 对象是可哈希的，可以用作字典或集合的键。
- `collections.Counter` 实现多重集合。

5.5 栈

栈是支持快速后进先出的对象集。与列表或数组不同，栈通常不允许随机访问其包含的对象。插入和删除操作通常也称为压栈和出栈。

栈数据结构的一个现实类比是盘子：

新的盘子将添加到栈的顶部。而且因为这些盘子又贵又重，只有最上面的盘子可以移动（后进先出）。想要拿到在栈中较低的位置的盘子，必须首先一一删除最上面的盘子。

栈和队列相似。它们都是对象的线性集合，区别在于访问的顺序：

通过队列，你可以删除最先添加的元素（先进先出或FIFO）；但是在栈中，你总是删除最近一次添加的元素（后进先出或LIFO）。

在性能方面，恰当的栈实现插入和删除操作的时间复杂度是O(1)。

栈在算法中有广泛的用途，例如在语言解析和运行时内存管理（“调用栈”）中。使用栈的一个简短优美的算法是在树或图数据结构上进行的深度优先遍历（DFS）。

Python附带了几个栈实现，每个都有略有不同。现在，我们来比较一下他们的特征。

list 简单的内置栈

Python内置的列表类型是栈这种数据结构的一个很好实现，它压栈和出栈的平均时间复杂度是O(1)。

Python的列表在内部以动态数组的形式实现，这意味着在添加或删除元素时，有时需要调整元素的存储空间大小。列表分配了多余的存储空间，因此并非每次压栈或出栈都需要调整大小，结果就是平摊后，得到了O(1)的时间复杂度。

不利的一面是，与基于链表的实现相比，这会使他们的性能不太稳定，基于链表的实现在插入和删除时可以提供稳定的O(1)时间复杂度（类似于 `collections.deque`，请参见下文）。在另一方面，列表在对元素进行随机访问时，可以保证O(1)时间复杂度，这是一个额外的好处。

这是在你使用列表作为栈时应该注意的一个性能警告：

为了在插入和删除时获得的均摊的O(1)时间复杂度，新增元素必须使用 `append()` 方法将元素添加到列表的末尾，并使用 `pop()` 从尾部进行删除。为了获得最佳性能，基于Python列表的栈在添加元素时，应按照索引增大的方向进行，删除元素应按照索引减小的方向进行。

从前面添加和删除要慢得多，并且时间复杂度是O(n)，因为必须为新元素转移现有元素并腾出空间。这是你应该尽量避免低性能模式。

```

>>> s = []
>>> s.append('eat')
>>> s.append('sleep')
>>> s.append('code')

```

```

>>> s

['eat', 'sleep', 'code']
>>> s.pop()
'code'
>>> s.pop()
'sleep'
>>> s.pop()
'eat'

>>> s.pop()
IndexError: "pop from empty list"

```

collections.deque 快速且健壮的栈

deque 类实现了一个双端队列，该队列支持用O(1)时间中从任意一端添加和删除元素（未摊销）。因为双端队列从任一端用添加和删除元素拥有同样的性能，它们既可以用作队列，也可以用作栈。

Python的双端队列通过双向链表的形式实现，这为他们提供了出色且一致的插入和删除性能，但随机访问栈中间的元素性能很差，时间复杂度为O(n)。

总体而言，如果你正在从Python标准库中寻找一个栈，那么 collections.deque 是一个不错的选择，它具有链表实现的性能特征。

```

>>> from collections import deque
>>> s = deque()
>>> s.append('eat')
>>> s.append('sleep')

>>> s.append('code')
>>> s
deque(['eat', 'sleep', 'code'])
>>> s.pop()
'code'
>>> s.pop()
'sleep'
>>> s.pop()
'eat'
>>> s.pop()
IndexError: "pop from an empty deque"

```

queue.LifoQueue ——并行计算时提供锁

Python标准库中的这个栈提供了锁定语义以支持多个生产者和消费者的并发操作。

除了 LifoQueue，queue 模块还包含其他几个类，这些队列可用于多生产者/多消费者时的并行计算。

根据你的场景，锁定语义可能会有所帮助，也有可能会产生不必要的开销。在这种情况下，你最好使用 list 或 deque 作为通用栈。

```

>>> from queue import LifoQueue
>>> s = LifoQueue()
>>> s.put('eat')
>>> s.put('sleep')
>>> s.put('code')
>>> s
<queue.LifoQueue object at 0x108298dd8>
>>> s.get()
'code'
>>> s.get()
'sleep'
>>> s.get()
'eat'
>>> s.get_nowait()
queue.Empty
>>> s.get()
# Blocks / waits forever...

```

Python中栈实现方法的比较

你看，Python附带了一些栈的实现。在性能和使用方面，它们都略有不同。

如果你没有并行处理的需求（或者不想手动处理加锁和解锁），你的可在内置列表类型或 `collections.deque` 中选择。区别在于底层使用的数据结构和易用性：

列表底层由动态数组实现，因此非常适合随机访问，但在添加或删除元素时偶尔需要调整大小。列表会分配多余的后备存储，因此并非每次压栈或出栈操作都需要调整大小，因此这些操作的均摊时间复杂度是O(1)。但你确实需要注意只使用 `append()` 和 `pop()` 进行插入和删除操作。否则，性能会降低到O(n)。

`collections.deque` 由双向链表实现，该链表优化了两端的追加和删除性能，并为这些操作提供了稳定的O(1)时间复杂度。双端队列不仅添加或删除元素时性能更加稳定，并且也不用担心在错误的一端添加或删除元素。

总而言之，如果要在Python中使用栈（LIFO），`collections.deque` 是一个不错的选择。

重点

- Python附带了几种栈的实现，性能和使用方面略有不同。
- `collections.deque` 提供了安全，快速的通用栈实现。
- 内置列表类型可以用作栈，但要注意，为了避免性能下降，只通过 `append()` 和 `pop()` 添加和删除元素。

5.6 队列（先进先出）

在本节中，你将了解如何仅使用Python标准库中内置的数据类型和类来实现FIFO队列。但首先，让我们回顾一下队列是什么：

队列是支持快速先进先出的数据集（FIFO）。插入和删除操作有时称为入队和出队。与列表或数组不同，队列通常不允许随机访问。

这是先进先出队列的一个真实类比：

想象一下，有一群程序员在等着拿他们的会议徽章。新来的人将添加到队列的后面，新人们进入会议场地并“排队”领取到他们的徽章。出队（服务）发生在队列前端，当程序员收到他们的徽章后就会离开队列。

记忆队列数据结构特征的另一种方法将其视为管道：

新物品（水分子，乒乓球等）从一端放入然后到达另一端，在这边你或其他人将其移开。当物品在排队（实心金属管）时，你无法访问它们。唯一的与队列中的元素进行交互的方法是在队列后面添加新的元素（入队）或在队列前面删除元素（出队）。

队列类似于栈，它们之间的区别在于如何删除项目：

在队列中，你删除的是最先添加的元素（先进先出或FIFO）；但是在栈中，你删除的是最近添加的元素（后进先出或LIFO）。

在性能方面，队列用于插入和删除的时间复杂度是O(1)。这是在队列上执行的两个主要操作，在正确实现的队列中，这两个操作应该要很快。

队列在算法中有广泛的应用，通常可以帮助解决调度和并行编程问题。一个简洁的使用队列的算法是在树或图数据结构上进行的广度优先搜索（BFS）。

调度算法通常在内部使用优先级队列。这些都是特殊的队列：优先级队列通过优先级获取元素，而不是通过元素插入时间。各个元素的优先级由队列根据其键确定。在下一章中，我们将仔细研究优先级队列及其在Python中的实现。

但是，常规队列不会重新排序所包含的元素。就像在管道示例中一样，你从队列中获取到的元素顺序和添加到队列的顺序相同。

Python附带了几个队列实现，每个实现都略有不同。让我们回顾一下。

list ——超级慢的队列

可以将列表用作队列，但考虑到性能问题，这是个糟糕的方案。用列表做队列会很慢，原因是在列表开头插入或删除元素需要将所有其他元素进行移动，时间复杂度是O(n)。

因此，在Python中，我不建议将列表用作临时队列（除非你只处理少量元素）。

```
>>> q = []
>>> q.append('eat')
>>> q.append('sleep')
>>> q.append('code')
>>> q
```

```
[ 'eat', 'sleep', 'code']
# Careful: This is slow!
>>> q.pop(0)
'eat'
```

collections.deque ——快速和健壮的队列

deque 类实现了一个双端队列，该队列从任一端添加和删除元素的时间复杂度都是O(1)（未摊销）。因为双端队列从任一端添加和删除元素的性能都同样好，所以它既可以用作队列，也可以用作栈。

Python的双端队列通过双向链表实现。这为插入和删除元素提供了出色且一致的性能，但随机访问性能较差，时间复杂度是O(n)。

因此，如果你要在Python的标准库中找一个队列的实现，则 `collections.deque` 是一个不错的默认选择

```
>>> from collections import deque
>>> q = deque()
>>> q.append('eat')
>>> q.append('sleep')
>>> q.append('code')
>>> q
deque(['eat', 'sleep', 'code'])
>>> q.popleft()
'eat'
>>> q.popleft()
'sleep'
>>> q.popleft()
'code'
>>> q.popleft()
IndexError: "pop from an empty deque"
```

queue.Queue ——在并行计算中提供锁定语义的队列

Python标准库中的这个队列提供了锁定语义以支持多个生产者和消费者的并发操作。

`queue` 模块还包含其他几个类，这些队列可用于多生产者/多消费者时的并行计算。

根据你的场景，锁定语义可能会有所帮助，也有可能会产生不必要的开销。在这种情况下，你最好使用 `dequeue` 作为队列。

```
>>> from queue import Queue
>>> q = Queue()
>>> q.put('eat')
>>> q.put('sleep')
>>> q.put('code')
>>> q
<queue.Queue object at 0x1070f5b38>
>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'
>>> q.get_nowait()
queue.Empty
>>> q.get()
# Blocks / waits forever...
```

multiprocessing.Queue ——共享任务的队列

这是一个共享任务的队列，允许排队的元素被多个worker并行处理，因为全局解释器锁（GIL）阻止了在单个进程中进行并行执行，所以通过进程实现并行在Python中很流行。

作为专门用于在多个进程之间共享数据的队列，使用 `multiprocessing.Queue` 可以很容易地在多个进程之间分配工作。这种队列可以在进程间存储和传输任何可以pickle的对象。

```
>>> from multiprocessing import Queue
>>> q = Queue()
```

```

>>> q.put('eat')
>>> q.put('sleep')
>>> q.put('code')
>>> q
<multiprocessing.queues.Queue object at 0x1081c12b0>
>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'
>>> q.get()
# Blocks / waits forever...

```

重点

- Python包含了几个队列实现。
- 列表对象可以用作队列，但通常不推荐这么做，因为它的性能很差。
- 如果你需要并行处理，`collections.deque` 是实现FIFO队列绝佳选择。它提供了良好的队列应有的性能，也可以用作栈（LIFO队列）

5.7 优先级队列

优先级队列是一种容器数据结构，它用于管理一组带有排序键（例如，权重值）的元素，可以快速访问数据集中最小或最大的key对应的元素。

你可以将优先级队列视为修改过的队列：它不是通过插入时间来获取下一个元素，而是获取优先级最高的元素。各个元素的优先级取决于应用于其键的顺序。

优先级队列通常用于处理调度问题，例如，优先处理优先级较高的任务。考虑一下操作系统的任务调度：

理想情况下，系统上的高优先级任务（例如玩实时游戏）应优先于低优先级任务（例如，在后台运行的任务）。通过将待处理的任务放在优先级队列，并根据紧急程度排定优先级，调度程序可以快速选择优先级最高的任务进行执行。

在本章中，你将看到Python标准库中优先级队列的一些实现方式。每个实现有其自己的优点和缺点，但是在我看来，通常情况下有一个很好的默认选项。让我们找出它是哪一个是。

list ——手动维护一个排序的队列

你可以使用排序的列表快速获取和删除列表中最小或最大的元素。缺点是将新元素插入到列表中很缓慢，时间复杂度是O(n)。

虽然可以使用标准库中的`bisect.insort` 在O(logn)时间内找到插入点，但是缓慢的插入操作始终占主导地位。

通过追加元素到列表并重新排序来维护顺序至少需要O(nlogn)的时间。另一个缺点是，在插入新元素时，你必须手动对列表进行重新排序。忽略此步骤很容易引入bug，而这始终是程序员的责任。

因此，我认为排序列表仅适合插入操作很少的优先级队列。

```

q = []
q.append((2, 'code'))
q.append((1, 'eat'))
q.append((3, 'sleep'))
# NOTE: Remember to re-sort every time
# a new element is inserted, or use
# bisect.insort().
q.sort(reverse=True)
while q:
    next_item = q.pop()
    print(next_item)
# Result:
# (1, 'eat')
# (2, 'code')
# (3, 'sleep')

```

heap ——基于列表的堆

这是一个基于队列的二进制堆实现，它支持在O(logn)时间内插入和提取最小元素。

该模块是在Python中实现优先级队列的不错选择。由于技术上 heapq 仅提供最小堆实现，因此必须采取额外的步骤来确保排序稳定性和其他通用优先级队列具有的功能。

```
import heapq
q = []
heapq.heappush(q, (2, 'code'))
heapq.heappush(q, (1, 'eat'))
heapq.heappush(q, (3, 'sleep'))
while q:
    next_item = heapq.heappop(q)
    print(next_item)
# Result:
# (1, 'eat')
# (2, 'code')
# (3, 'sleep')
```

queue.PriorityQueue —— 优美的优先级队列

这个优先级队列在底层通过 heapq 实现，并且和堆具有同样的时间复杂度和空间复杂度。

区别在于 PriorityQueue 提供锁定语义，支持多个并发的生产者和消费者。

根据你的使用场景，这可能会有所帮助，也可能拖慢你的程序。无论如何，你可能更喜欢 PriorityQueue 基于类的接口而不是 heapq 提供的基于函数的接口。

```
from queue import PriorityQueue
q = PriorityQueue()
q.put((2, 'code'))
q.put((1, 'eat'))
q.put((3, 'sleep'))
while not q.empty():
    next_item = q.get()
    print(next_item)
# Result:
# (1, 'eat')
# (2, 'code')
# (3, 'sleep')
```

重点

- Python包含了几种优先级队列实现供你使用。
- queue.PriorityQueue 从所有的选择中脱颖而出，它具有表现优美的面向对象接口和一个明确的名称。它应该是你的首选。
- 如果你想避免 queue.PriorityQueue 的锁开销，直接使用 heapq 模块也是一个不错的选择。

source/python_tricks/6_循环和迭代.md

第六章 循环和迭代

6.1 编写python循环

在最近转Python的开发人中，找出具有C语言背景的人的最简单方法之一就是看他们如何写循环。

例如，如下代码段就是有人在尝试用C或Java的风格写Python的一个例子：

```
my_items = ['a', 'b', 'c']
i = 0
while i < len(my_items):
    print(my_items[i])
    i += 1
```

现在，这段代码有那些地方不符合Python风格？ 两个点：

首先，它手动维护索引 `i` ——将其初始化为零，然后在每次循环迭代时增加它。

其次，为了确定迭代次数，它使用 `len()` 来获取 `my_items` 容器的大小。

在Python中，你可以编写自动处理这两个问题的循环。充分利用这一点是一个好主意。例如，如果你不必处理索引，那么就很难编写意外的无限循环。这也使代码更简洁，因此更具可读性。

为了重构这个代码示例，我将从删除手动更新索引的代码开始。在Python中，一个好的方法是使用for循环。使用内置的 `range()` 函数，我可以自动生成索引：

```
>>> range(len(my_items))
range(0, 3)
>>> list(range(0, 3))
[0, 1, 2]
```

`range` 类型表示一个不可变的数字序列。相对于列表来说，它只需要很小的内存。`range` 对象实际上并不存储代表数字序列的每个值，相反，它是一个迭代器，会实时计算序列值。

因此，与其在每次循环迭代中手动增加`i`，不如利用 `range()` 函数这么写：

```
for i in range(len(my_items)):
    print(my_items[i])
```

这钟写法更好。但是，它仍然不是很Pythonic，它更像Java式的迭代结构，而不是Python的循环。当你看到使用 `range(len(...))` 遍历容器的代码时，你通常可以进一步简化和改进它。

正如我提过的，在Python中，for循环实际上是for-each循环，可以直接遍历容器或序列中的项目，而无需按索引查找它们。我可以用它来简化这个循环：

```
for item in my_items:
    print(item)
```

我认为这种方案是相当Pythonic的。它使用了几种先进的Python功能，但仍保持简洁美观，几乎就像编程教科书中的伪代码一样。请注意一下这个循环是如何不再跟踪容器的大小并且不再使用索引访问元素。

容器本身现在负责产出元素，所以元素才可以被处理。如果容器有序，则结果也将有序。如果容器无序，结果也将无序，但循环仍将覆盖所有元素。

当然，你不可能一直可以像这样重写你的循环。例如，如果你需要元素的索引时，该怎么办？

在不使用 `range(len(...))` 这种写法时，也可以编写循环来维护索引。内置的 `enumerate()` 函数可帮助你使这些循环变得美观且符合 Python 风格：

```
>>> for i, item in enumerate(my_items):
...     print(f'{i}: {item}')
0: a
1: b
2: c
```

你看，Python 中的迭代器可以返回多个值。他们可以返回具有任意数量元素的元组，然后可以在 for 语句中对元组进行拆包。

这是非常强大的。例如，你可以使用相同的方法同时遍历字典的键和值：

```
>>> emails = {
...     'Bob': 'bob@example.com',
...     'Alice': 'alice@example.com',
... }
>>> for name, email in emails.items():
...     print(f'{name} -> {email}')
'Bob -> bob@example.com'
'Alice -> alice@example.com'
```

我想再举一个例子给你看。如果你就是需要一个 C 风格的循环，该怎么办？例如，如果你必须控制索引的步长？想象一下，你从以下 Java 循环开始：

```
for (int i = a; i < n; i += s) {
// ...
}
```

这种模式如何转换到 Python？使用 `range()` 函数可以解决这个问题——它接受可选参数来控制循环的起始值(a)，结束值(n)和步长(s)。

因此，我们的 Java 循环可以转换成像下面的 Python 语句：

```
for i in range(a, n, s):
# ...
```

重点

- C 风格的循环是不符合 Python 风格的。尽可能避免手动管理循环的索引和停止条件。
- Python 的 for 循环实际上是 for-each 循环，它可以直接从容器或序列对元素进行迭代。

6.2 解析

我最喜欢的 Python 特性之一就是列表解析。他们乍一看似乎有些不可思议，但是当你将它们分解时，它们实际上是一个非常简单的结构。

理解列表解析的关键在于，它们只是用了一种更简洁和紧凑的语法对一个数据集进行 for 循环。

这有时也称为语法糖，是让我们这些 Python 程序员更轻松一点的常用功能的简写。以下以列表解析为例：

```
>>> squares = [x * x for x in range(10)]
```

它计算从零到九的所有整数的平方：

```
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

如果你想使用普通的 for 循环构建相同的列表，则可能需要这样写：

```
>>> squares = []
>>> for x in range(10):
```

```
...     squares.append(x * x)
```

那是一个非常简单的循环，对不对？如果你回过头来比较列表解析的示例和for循环示例，找出共同点，那么最终会总结出一些模式。通过概括一些常见的结构，你最终将总结出类似于以下的模板：

```
values = [expression for item in collection]
```

上面的列表解析模板等效于下面的普通for循环：

```
values = []
for item in collection:
    values.append(expression)
```

在这里，我们首先创建一个新的列表实例以接收输出值。然后，我们遍历容器中的所有元素，用任意表达式对每个元素进行转换，然后将结果添加到输出列表。

这是一个千篇一律的模式，你可以将其应用于许多for循环以将其转化为列表解析，反之亦然。现在，我们需要在这个模板中添加一个更有用的部分，那就是根据条件过滤元素。

列表解析可以基于一些任意条件来过滤值，这些条件决定结果值是否成为输出列表的一部分。这是一个例子：

```
>>> even_squares = [x * x for x in range(10)
                   if x % 2 == 0]
```

这个列表解析将计算从零到九的整数中所有偶数的平方。此处使用的模(%)运算符返回一个数字除以另一个后的余数。在此示例中，我们将使用它来测试数字是否为偶数。结果如下：

```
>>> even_squares
[0, 4, 16, 36, 64]
```

与第一个示例类似，这种列表解析也可以转换为等效的for循环：

```
even_squares = []
for x in range(10):
    if x % 2 == 0:
        even_squares.append(x * x)
```

让我们尝试概括一下列表解析到for循环的转换模式。这次我们要在模板中添加一个过滤条件，因此我们可以决定哪个元素会输出到最终的结果列表。这是更新的列表解析模板：

```
values = [expression
          for item in collection
          if condition]
```

同样，我们可以将列表解析转换为for循环：

```
values = []
for item in collection:
    if condition:
        values.append(expression)
```

再一次，这是一个简单的转换——我们只是应用了更新后的模式。我希望这能消除一些与列表解析如何工作相关的“魔术”。这是一个所有Python程序员都应该知道如何使用的有效工具。

在继续之前，我想指出Python不仅仅支持列表解析，还有类似的语法糖用于支持集合和字典。集合解析如下：

```
>>> { x * x for x in range(-9, 10) }
set([64, 1, 36, 0, 49, 9, 16, 81, 25, 4])
```

与列表保留了元素的顺序不同，Python集是无序的。因此，将元素添加到集合中时，结果的顺序将或多或少“随机”过。

这是字典解析：

```
>>> { x: x * x for x in range(5) }
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

两者在实践中都是有用的工具。不过，对于Python的解析有一个警告：随着你越来越熟练地使用它们，编写难以阅读的代码会变得越来越容易。一不小心就可能不得不处理可怕的列表，集合和字典。请记住，过犹不及。

经过一番苦恼之后，我个人将底线定在一层嵌套的列表解析。我现在超过一层嵌套之后，大多数情况下使用for循环效果会更好（“更易读”和“更易于维护”）。

重点

- 解析表达式是Python中的关键功能。理解并应用它们将使你的代码更加Pythonic。
- 解析表达式只是for循环的语法糖。了解了模式之后，你会对解析表达式有更深的理解。
- 列表解析之外还有其它的解析表达式。

6.3 列表切片技巧

Python的列表对象具有一项巧妙的功能，称为切片。它是方括号索引语法的扩展。切片通常用于访问有序集合中一个范围内的元素。例如，你可以将一个大型列表对象切成多个较小的子列表。

这是一个例子。切片使用常见的“[]”语法，使用 [start: stop: step] 模式：

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst
[1, 2, 3, 4, 5]
# lst[start:end:step]
>>> lst[1:3:1]
[2, 3]
```

添加 [1:3:1] 索引返回原始列表的一部分，这个部分从索引1开始到索引2结束，步长为一个元素。为了避免边界条件错误，要记住上限永远不包含在内。这就是为什么 [1:3:1] 切片会返回 [2,3] 子列表的原因。如果你忽略步长，则默认为一：

```
>>> lst[1:3]
[2, 3]
```

你还可以使用step参数执行其他有趣的操作。例如，你可以从原始列表每隔1个元素取一个以创建子列表：

```
>>> lst[::-2]
[1, 3, 5]
```

你刚刚看到切片步长如何用于从原始列表中每隔1个取1个元素。好吧，还有更多：如果你使用 [::-1] 切片，你将获得一个原始列表的副本，只是顺序相反：

```
>>> numbers[::-1]
[5, 4, 3, 2, 1]
```

我们使用(:)要求Python提供一个完整的列表，然后通过将步长设置为-1，从后到前排遍历所有元素。好整洁，但在大多数情况下，我仍然坚持使用 `list.reverse()` 和内置的 `reversed()` 函数转列表。

这是另一个列表切片技巧：你可以使用:操作符清除列表中的所有元素，而不会销毁列表对象本身。

当你需要清除程序中具有其他引用指向的列表时，这非常有用。在这种情况下，你通常不能仅仅通过用新列表对象替换旧列表来清空列表，因为引用不会更新：

```
>>> lst = [1, 2, 3, 4, 5]
>>> del lst[:]
```

```
>>> lst
[]
```

你看，这将删除lst中的所有元素，但列表象本身完好无损。在Python 3中，你还可以使用 `lst.clear()` 实现同样的功能，这可能是更具可读性的模式。但是，Python 2无法使用 `clear()`。

除了清除列表，你还可以使用`:`操作符在不创建新列表的前提下替换所有元素。这是一个清除列表，然后手动重新填充它例子：

```
>>> original_lst = lst
>>> lst[:] = [7, 8, 9]
>>> lst
[7, 8, 9]
>>> original_lst
[7, 8, 9]
>>> original_lst is lst
True
```

前面的代码示例替换了lst中的所有元素，但没有销毁并重新创建列表本身。之前对列表对象的引用仍然有效。

`:`操作符的另一个用途是创建浅拷贝：

```
>>> copied_lst = lst[:]
>>> copied_lst
[7, 8, 9]
>>> copied_lst is lst
False
```

创建浅拷贝意味着仅元素的结构被复制，而不是元素本身。列表的两个副本共享各个元素的相同实例。

如果你需要复制包括元素在内的所有内容，那么你需要创建列表的深拷贝。Python内置的 `copy` 模块可以实现这个功能。

重点

- `:`操作符不仅对选择子列表有用，它也可以用于清除，反转，并复制列表。
- 但要小心——这个功能对许多Python开发人员有点神秘，使用它可能会降低你的代码的可维护性。

6.4 漂亮的迭代器

与许多其他编程语言的语法相比，Python的语法很简洁。让我们以 `for-in` 循环为例。这样简洁的循环，就好像是一个英语句子：

```
numbers = [1, 2, 3]
for n in numbers:
    print(n)
```

Python这样优雅的循环是如何实现的呢？循环如何从正在循环的对象中获取每个元素？在你自己的Python对象中如何才能支持相同的编程风格？

你可以在Python的迭代器协议中找到这些问题的答案：支持 `__iter__` 和 `__next__` 双下划线方法的对象会自动支持`for-in`循环。

让我们一步步来。和装饰器一样，迭代器及其的相关技术乍一看非常神秘和复杂。因此，我们将简化它们。

在本章中，你将看到如何编写多个支持迭代器协议的Python类。它们是“非魔术”示例，这是你建立并加深理解的一些测试实现。

首先，我们排除任何不必要的环节，把重点放在介绍Python 3中迭代器的核心机制上，因此你可以清楚地看到迭代器在底层怎么运行。

我会将每个示例都与我们刚开始提到的`for-in`循环问题联系在一起。并且，在本章的最后，我们将讨论一些迭代器在Python 2和Python 3之间的差异。

准备好了吗？让我们开始吧！

永远迭代

首先，我们从编写一个演示最基本的迭代器协议的类开始。我在这里使用的示例看起来可能与你在其他迭代器教程中看到的示例不同。我认为通过这种方式可以让你更了解迭代器在Python中的工作方式。

在接下来的几段中，我们将实现一个名为 Repeater 的类，这个类可以使用for-in循环进行迭代，如下所示：

```
repeater = Repeater('Hello')
for item in repeater:
    print(item)
```

顾名思义，这个Repeater类的实例在迭代时将重复返回单个值。所以上面的例子将永远在控制台上输出字符串“Hello”。首先，我们先定义并实现 Repeater：

```
class Repeater:
    def __init__(self, value):
        self.value = value

    def __iter__(self):
        return RepeaterIterator(self)
```

第一次看时，Repeater 看起来像是标准的Python类。但是注意，它还包含了 `__iter__` 方法。

我们从 `__iter__` 创建并返回的 `RepeaterIterator` 对象是什么？这是为了让for-in迭代示例可以运行需要定义的一个帮助类：

```
class RepeaterIterator:
    def __init__(self, source):
        self.source = source

    def __next__(self):
        return self.source.value
```

同样，`RepeaterIterator` 看起来像一个简单的Python类，但你可能要注意以下两件事：

1. 在 `__init__` 方法中，我们将每个 `RepeaterIterator` 都链接到创建它的 `Repeater` 对象。这样我们可以保持被迭代的“源”对象。
2. 在 `RepeaterIterator.__ next__` 中，我们返回到“源”（`Repeater` 实例）并返回它关联的值。

在此代码示例中，`Repeater` 和 `RepeaterIterator` 一起支持了Python的迭代器协议。我们定义的两个下划线方法 `__iter__` 和 `__next__` 是实现Python可迭代对象的关键。

我们接下来会仔细研究这两种方法，我们会对已有的代码进行一些实验，然后看看他们怎么运行。让我们确认一下这个两个类确实使 `Repeater` 对象可以通过for-in循环迭代。我们将首先创建一个 `Repeater` 实例，该实例将一直返回字符串“Hello”：

```
>>> repeater = Repeater('Hello')
```

现在，我们尝试使用for-in循环。当你运行以下代码时会发生什么？

```
>>> for item in repeater:
...     print(item)
```

你会看到很多“Hello”输出到屏幕上。`Repeater`一直输出同样的字符串到屏幕，并且这个循环永远不会结束。我们的程序注定要永远打印'Hello'到控制台：

```
Hello
Hello
Hello
Hello
Hello
...
...
```

恭喜——你刚刚用Python写了一个可以运行的迭代器，并且和for-in循环一起使用。循环可能还没有结束……但是到目前为止，没有什么问题。

接下来，我们将梳理这个示例，以了解 `__iter__` 和 `__next__` 方法如何让Python对象可迭代。

提示：如果你运行了最后一个示例，现在想要停止它，请按几次 `Ctrl + C` 以摆脱无限循环。

for-in循环在Python中如何工作？

很明显我们的 `Repeater` 类支持迭代器协议，我们运行了一个for-in循环来证明它：

```
repeater = Repeater('Hello')
for item in repeater:
    print(item)
```

现在，这个for-in循环在底层做了什么？它是如何与 `Repeater` 对象通信并从中获取新元素的？

为了消除某些“魔法”，我们可以将此循环扩展为一个长一点的版本，两者具有相同的结果：

```
repeater = Repeater('Hello')
iterator = repeater.__iter__()
while True:
    item = iterator.__next__()
    print(item)
```

你看，`for-in`只是一个简单 `while` 循环的语法糖：

- 它首先通过调用 `__iter__` 方法让 `repeater` 对象准备进行迭代。这返回了实际的迭代器对象。
- 之后，循环反复调用迭代器对象的 `__next__` 方法从中检索值。

如果你使用过数据库游标，这个思维模型看起来会很熟悉：我们首先初始化游标并准备进行读取，然后我们可以根据需要将数据读取到局部变量中，一次一个元素。

因为正在处理中的元素不超过一个，所以这种方法具有很高的内存使用效率。我们的 `Repeater` 类提供了一个无限的元素序列，我们可以对其进行迭代。用Python列表模拟同样的事情是不可能的——我们不可能一开始就创建一个具有无限元素的列表。这使得迭代器成为一个非常强大的概念。

用更抽象的术语讲，迭代器提供了一个通用接口，它可让你在与容器的内部结构完全隔离的情况下，处理容器的每个元素。

无论你要处理的是列表，字典，无限序列（比如 `Repeater` 类提供的这种）或其它序列类型——所有这些都只是实现细节。每个这种对象，都可以使用迭代器以相同的方式进行遍历。

你看，在Python中`for-in`循环并没有什么特别的。如果你探究一下底层的原理，归根结底是在正确的时间调用正确的双下划线方法。实际上，你可以在Python解释器会话中手动“模拟”循环如何使用迭代器协议：

```
>>> repeater = Repeater('Hello')
>>> iterator = iter(repeater)
>>> next(iterator)
'Hello'
>>> next(iterator)
'Hello'
>>> next(iterator)
'Hello'
...
...
```

这输出了相同的结果——无限的 `Hello` 流。每次你调用 `next()` 时，迭代器都输出相同的问题。

顺便说一下，我在这里用Python内置函数 `iter()` 和 `next()` 取代了对 `__iter__` 和 `__next__` 的调用。

在内部，这些内置函数调用相同的双下划线方法，但是通过提供一个迭代器协议的整洁“门面”，它们让代码更整洁，更易读。

Python还为其它方法提供了快捷方式。例如，`len(x)` 是调用 `x.__len__` 的快捷方式。相似地，调用 `iter(x)` 最终会调用 `x.__iter__`，调用 `next(x)` 最终会调用 `x.__next__`。

通常，最好使用内置函数，而不是直接调用实现协议的双下划线方法。它会使代码更易于阅读。

一个简单的迭代类

到目前为止，我们的迭代器示例由 `Repeater` 和 `RepeaterIterator` 两个单独的类组成。他们直接对应于Python的迭代器协议的两个阶段：

首先，通过调用 `iter()` 创建迭代器对象，然后通过 `next()` 从中获取值。

很多时候，这两项职责可以由一个类承担。在编写基于类的迭代器时，这样做可以减少所需的代码量。

我不选择在本章的第一个示例这么做，是因为它让迭代器协议背后的心理模型不再清晰。但是现在你已经了解了如何用更长，更复杂的方法编写基于类的迭代器，接下来让我们花一点时间来简化一下我们目前的代码。

还记得为什么我们需要 `RepeaterIterator` 类吗？我们需要它来承载 `_next__` 方法以从迭代器中获取新值。但是 `_next__` 在哪里定义并不重要。在迭代器协议中，重要的是 `_iter__` 要返回带有 `_next__` 方法的对象。

所以这里有个想法：`RepeaterIterator` 一直返回相同的值，不必跟踪任何内部状态。如果我们直接将 `_next__` 方法添加到 `Repeater` 类中会怎么样？

这样，我们可以完全摆脱 `RepeaterIterator`，并用单个Python类实现一个可迭代的对象。让我们试试！我们新简化后的新迭代器示例如下：

```
class Repeater:
    def __init__(self, value):
        self.value = value

    def __iter__(self):
        return self

    def __next__(self):
        return self.value
```

我们从10行代码的两个单独的类简化到了7行代码的一个类。我们简化的实现仍然很好地支持迭代器协议：

```
>>> repeater = Repeater('Hello')
>>> for item in repeater:
...     print(item)
Hello
Hello
Hello
...
...
```

简化这样基于类的迭代器通常很有意义。实际上，大多数Python的迭代器教程都是以这种方式开始的。但是我总是觉得从一开始就用一个类来解释迭代器没有揭示迭代器协议的底层的原理，因此使其难以理解。

谁想一直迭代

现在，你应该对Python迭代器的工作方式有了很好的理解。但是到目前为止，我们仅实现了一个一直迭代的迭代器。

显然，无限重复并不是迭代器的主要使用场景。实际上，当你从头到尾回顾整个过程时，在本章中，我使用以下代码段作为开始的示例：

```
numbers = [1, 2, 3]
for n in numbers:
    print(n)
```

你会期望上面的代码打印数字1、2和3，然后停下来。你不会期望它在你的终端一直打印“3”，直到你在慌乱中使用了 `Ctrl + C` ...

因此，是时候看看怎么编写最后能够停止生成新值，而不是永远迭代下去的迭代器了，因为那才是我们在`for-in`循环中使用迭代器时典型的表现。

现在，我们来编写另一个迭代器类，称为 `BoundedRepeater`。它与之前的 `Repeater` 示例相似，但是这次我们希望它在预设的重复次数后停止。

让我们考虑一下。我们如何做到这一点？迭代器如何在没有元素要迭代时发出信号？也许你在想，“嗯，我们可以在 `_next__` 方法返回 `None`。”

那不是一个坏主意，但问题是，如果我们希望某些迭代器能够返回 `None` 做为可接受的值时应该怎么办？

让我们看看其他迭代器是怎么解决这个问题的。我将构造一个简单的容器，一个包含一些元素的列表，然后我将对其进行迭代，直到用完所有元素以查看发生了什么事情：

```
>>> my_list = [1, 2, 3]
>>> iterator = iter(my_list)

>>> next(iterator)
1
>>> next(iterator)
2
>>> next(iterator)
3
```

注意！我们已经使用了列表中的仅有的三个可用元素。看看如果再次在迭代器上调用 `next` 会发生什么：

```
>>> next(iterator)
StopIteration
```

啊哈！它会抛出 `StopIteration` 异常，表明我们已经耗尽了迭代器中所有可用的值。

没错：迭代器使用异常来构造控制流。Python迭代器通过抛出内置 `StopIteration` 异常，表示迭代结束。

如果我不断要求迭代器提供更多值，它将不断抛出 `StopIteration` 异常，以表示没有更多值可迭代了：

```
>>> next(iterator)
StopIteration
>>> next(iterator)
StopIteration
...
...
```

一旦耗尽，Python迭代器通常无法“重置”，他们应该在每次调用 `next()` 时抛出 `StopIteration` 异常。如果要重新进行迭代，需要使用 `iter()` 函数构造一个新的迭代器对象。

现在我们知道编写 `BoundedRepeater` 类所需的一切了，它会在设定的重复次数后停止迭代：

```
class BoundedRepeater:
    def __init__(self, value, max_repeats):
        self.value = value
        self.max_repeats = max_repeats
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.max_repeats:
            raise StopIteration
        self.count += 1
        return self.value
```

这给出了我们想要的结果。迭代器在迭代了 `max_repeats` 参数定义的迭代次数后停止了：

```
>>> repeater = BoundedRepeater('Hello', 3)
>>> for item in repeater:
    print(item)
Hello
Hello
Hello
```

如果我们去掉语法糖，重写这个for-in循环示例，我们得到了下面展开后的代码片段：

```
repeater = BoundedRepeater('Hello', 3)
iterator = iter(repeater)
while True:
    try:
```

```

item = next(iterator)
except StopIteration:
    break
print(item)

```

每次在此循环中调用 `next()` 时，我们都会检查 `StopIteration` 异常，并在必要时中断 `while` 循环。

能够编写三行的for-in循环而不是八行的while循环是一个不错的改进。它使代码更容易阅读且更易于维护。这是Python中迭代器如此强大的另一个原因。

Python2.x 兼容

我在这里展示的所有代码示例都是用Python 3编写的。在实现基于类的迭代器时，Python 2和3之间的区别很小但却很重要：

- 在Python 3中，从迭代器获取下一个值的方法称为 `__next__`。
- 在Python 2中，相同的方法称为 `next`（不带下划线）。

如果你尝试编写应该在两种版本的Python上都可以使用的基于类的迭代器，这种命名差异可能会导致一些问题。幸运的是，你可以采用一种简单的方法来绕过这种差异。

这是InfiniteRepeater类的更新版本，适用于Python 2和Python 3：

```

class InfiniteRepeater(object):
    def __init__(self, value):
        self.value = value

    def __iter__(self):
        return self

    def __next__(self):
        return self.value

    # Python 2 compatibility:
    def next(self):
        return self.__next__()

```

为了使该迭代器类与Python 2兼容，我对其做了两个小改进：

首先，我添加了一个 `next` 方法，该方法仅调用原始的 `__next__` 并转发其返回值。这实际是现有的 `__next__` 实现的别名，这样Python 2就可以找到它。这样我们就可以支持两个版本的Python，同时所有实现细节仍然集中在一处。

其次，我修改了类定义，让类继承自 `object`，确保我们在Python 2上创建的是新式类。这与迭代器没有任何关系，但尽管如此这是一个好习惯。

重点

- 迭代器为Python对象提供了一个序列接口，它具有很高的内存使用效率，并且是Pythonic的。看漂亮的for-in循环！
- 为了支持迭代，对象需要提供 `__iter__` 和 `__next__` 双下划线方法来实现迭代器协议。
- 基于类的迭代器只是在Python中编写可迭代对象的一种方法。还有生成器和生成器表达式。

6.5 生成器是简化的迭代器

在关于迭代器的章节中，我们花了大量时间编写基于类的迭代器。从教育的观点来看，这并不是一个坏主意——但它也表明了编写一个迭代器类需要很多样板代码。说实话，作为一个“懒惰”的开发，我不喜欢乏味且重复的工作。

但是，迭代器在Python中是如此有用。它们让你写出漂亮的for-in循环，可使你的代码更Pythonic并且更加高效。如果有一种更简便的方法来编写这些迭代器.....

Python再一次为我们提供了一些语法糖，使编写迭代器变得更加容易。在本章中，你将看到如何使用生成器和 `yield` 关键字用更少的代码更快地编写迭代器。

无限的生成器

让我们再次来看一下我之前用来介绍迭代器概念的 Repeater 示例。它实现了基于类的迭代器，循环遍历无限序列。这是该类在其第二个（简化版）版本中的实现：

```
class Repeater:
    def __init__(self, value):
        self.value = value

    def __iter__(self):
        return self

    def __next__(self):
        return self.value
```

如果你在想，“对于这样一个简单的迭代器来说，代码有点多，”你这么想是对的。这个类的某些部分看起来是公式化的，好像它们在不同的基于类的迭代器中以完全相同的方式编写。

这是Python的生成器发挥作用的地方。如果我把这个迭代器改写成生成器的，它看起来像这样：

```
def repeater(value):
    while True:
        yield value
```

对于这里发生的事情来说这确实是一个很合适的思维模式。你看到，当在函数内部调用 `return` 语句时，它永久地将控制权传递回该函数的调用者。当 `yield` 被调用时，它同样会将控制权传回给函数的调用者——但是它只是暂时的。

而 `return` 语句丢弃了函数的局部状态，`yield` 语句挂起该函数并保留其局部状态。实际上，这意味着生成器函数的局部变量和执行状态只是暂时隐藏而不是完全扔掉。可以随时通过在生成器上调用 `next()` 恢复执行：

```
>>> iterator = repeater('Hi')
>>> next(iterator)
'Hi'
>>> next(iterator)
'Hi'
>>> next(iterator)
'Hi'
```

这使生成器与迭代器协议完全兼容。因此，我喜欢将它们看作是实现迭代器的语法糖。

你会发现，对于大多数类型的迭代器，编写生成器函数与定义一个基于类的迭代器相比，会更容易并且更具可读性。

可以停止的生成器

在本章中，我们再次通过编写一个无限生成器开始。现在，你可能想知道如何编写一个一段时间后停止产生值的生成器，而不是持续不断地运行。

记住，在基于类的迭代器中，我们能够通过手动抛出 `StopIteration` 异常发出结束信号。因为生成器与基于类的迭代器完全兼容，底层的实现原理也是一样的。

幸运的是，作为程序员，这次我们可以使用一个更好的接口。一旦控制流通过除了 `yield` 语句外的其它任何方式从生成器函数返回，生成器就停止生成值。这意味着你完全不必自己操心去抛出 `StopIteration`！这是一个例子：

```
def repeat_three_times(value):
    yield value
    yield value
    yield value
```

请注意，该生成器函数不包含任何循环。实际上，它非常简单，仅包含三个`yield`语句。如果`yield`暂时挂起该函数的执行并返回值给调用者，当我们到达这个生成器的末尾时会发生什么？让我们看看：

```
>>> for x in repeat_three_times('Hey there'):
...     print(x)
'Hey there'
'Hey there'
'Hey there'
```

如你所料，此生成器在三次迭代后停止生成新值。我们可以假设它是通过在执行到达末尾时，抛出 `StopIteration` 异常实现的。但是为了确定，让我们通过另一个实验来确认这一点：

```
>>> iterator = repeat_three_times('Hey there')
>>> next(iterator)
'Hey there'
>>> next(iterator)
'Hey there'
>>> next(iterator)
'Hey there'
>>> next(iterator)
StopIteration
>>> next(iterator)
StopIteration
```

该迭代器的行为与我们预期的一样。我们一到达生成器函数的末尾，它就不断抛出 `StopIteration` 来表示没有更多的值了。

让我们回到迭代器章节的另一个例子。`BoundedIterator` 类实现了一个迭代器，该迭代器仅重复设定的次数：

```
class BoundedRepeater:
    def __init__(self, value, max_repeats):
        self.value = value
        self.max_repeats = max_repeats
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.max_repeats:
            raise StopIteration
        self.count += 1
        return self.value
```

我们为什么不尝试将 `BoundedRepeater` 类重新实现为生成器函数呢。这是我的第一个实现：

```
def bounded_repeater(value, max_repeats):
    count = 0
    while True:
        if count >= max_repeats:
            return
        count += 1
        yield value
```

我故意使该函数中的 `while` 循环有点笨拙。我想要演示如何从生成器中调用 `return` 语句，如何使迭代因 `StopIteration` 异常而停止。我们很快就会清理并简化这个生成器函数，但是首先，让我们尝试一下到目前为止我们得到的生成器：

```
>>> for x in bounded_repeater('Hi', 4):
...     print(x)
'Hi'
'Hi'
'Hi'
'Hi'
```

现在我们有了一个生成器，它会在重复配置的次数之后停止生成值。它使用 `yield` 语句来传回值，直到最终命中 `return` 语句并停止迭代。

就像我向你保证的那样，我们可以进一步简化这个生成器。我们会利用Python在每个函数的末尾都添加了一个隐式的 `return None` 语句这个事实。这是我们最终的实现方式：

```
def bounded_repeater(value, max_repeats):
    for i in range(max_repeats):
        yield value
```

你可以随时去确认这个简化的生成器是否可以正常工作。考虑到所有因素，我们从 `BoundedRepeater` 类这个12行的实现转换为基于生成器的3行实现，两者功能完全相同。代码量减少了75%——还不赖！

你看，生成器可简化掉编写基于类的迭代器所需的大部分样板代码。他们可以使你的生活更加轻松，并让你编写更简洁，更短且更易于维护的迭代器。生成器函数在Python中是一个很棒的功能，你在你自己的程序中使用它们时不应该犹豫。

重点

- 生成器函数是用于编写支持迭代器协议对象的语法糖。生成器可简化掉编写基于类的迭代器时所需的大部分样板代码。
- `yield` 语句使你可以临时挂起生成器函数的执行并从中传递值。
- 控制流以 `yield` 以外的任何方式离开生成器函数后，生成器都会抛出 `StopIteration` 异常

6.6 生成器表达式

当我进一步了解Python的迭代器协议及其实现的不同方法后，我意识到“语法糖”是一个反复出现的主题。

你会看到，基于类的迭代器和生成器函数是相同设计模式的两个实现方式。

生成器函数为你提供了在你自己的代码中支持迭代器协议的捷径，并且避免了基于类的迭代器中的大量样板代码。借助一些专门的语法或语法糖，它们可以节省你的时间，让你的生活更加容易。

这是Python和其他编程语言中经常出现的主题。随着越来越多的开发人员在其程序中使用设计模式，语言创作者越来越有动力为其提供抽象和实现捷径。

这就是编程语言随着时间的推移而发展的方式——作为开发人员，我们将从中受益。我们开始使用越来越多的功能强大的模块，从而减少了工作量并使我们事倍功半。

在本书前面的章节，你看到了生成器如何为编写基于类的迭代器提供语法糖。我们在这一节要讲到的生成器表达式在外层又添加了另一层语法糖。

生成器表达式为你提供了更有效编写迭代器的快捷方式。使用看起来像列表解析的简单明了的语法，你可以用一行代码定义迭代器。这是一个例子：

```
iterator = ('Hello' for i in range(3))
```

迭代时，这个生成器表达式产生的值与我们在上一章中写的 `bounded_repeater` 生成器函数的值相同。在这里再次复习一下：

```
def bounded_repeater(value, max_repeats):
    for i in range(max_repeats):
        yield value
iterator = bounded_repeater('Hello', 3)
```

现在单行生成器表达式可以实现以前需要四行生成器函数或更长的基于类的迭代器才能实现的功能，这难道不令人震惊吗？

但是我要超越自己。确保我们使用生成器表达式定义的迭代器运行起来和我们的预期一致：

```
>>> iterator = ('Hello' for i in range(3))
>>> for x in iterator:
...     print(x)
'Hello'
'Hello'
'Hello'
```

这看起来不错！我们从一行的生成器表达式和 `bounded_repeater` 生成器函数似乎得到了相同的结果。

不过，有一个小警告：一旦生成器表达式被消耗完，它就再也无法重新启动或重用。所以在某些情况下使用生成器函数或基于类的迭代器有一些优势。

生成器表达式与列表解析

如你所知，生成器表达式与列表解析有点相似：

```
>>> listcomp = ['Hello' for i in range(3)]
>>> genexpr = ('Hello' for i in range(3))
```

但是，与列表解析不同，生成器表达式不会构造列表对象。相反，它们和基于类的迭代器或生成器函数一样“实时”生成值。

将生成器表达式赋值给变量后会得到可迭代的“生成器对象”：

```
>>> listcomp
['Hello', 'Hello', 'Hello']
>>> genexpr
<generator object <genexpr> at 0x1036c3200>
```

要访问生成器表达式产生的值，你需要调用 `next()`，这和其他迭代器是一样的：

```
>>> next(genexpr)
'Hello'
>>> next(genexpr)
'Hello'
>>> next(genexpr)
'Hello'
>>> next(genexpr)
StopIteration
```

另外，你也可以在生成器表达式上调用 `list()` 函数，以构造一个包含所有值的列表对象：

```
>>> genexpr = ('Hello' for i in range(3))
>>> list(genexpr)
['Hello', 'Hello', 'Hello']
```

当然，这只是一个玩具示例，展示了如何将生成器表达式“转换”（或与此相关的任何其他迭代器）为列表。如果你需要使用的是列表对象，通常只需从一开始就编写一个列表解析。

让我们仔细看看这个简单的生成器表达的句法结构。你开始看到的模式应该像这样：

```
genexpr = (expression for item in collection)
```

上面的生成器表达式“模板”对应于以下的生成器函数：

```
def generator():
    for item in collection:
        yield expression
```

就像列表推导一样，这为你提供了一种千篇一律的模式，你可以将其应用于许多生成器函数，然后将它们转换成简明的生成器表达式。

过滤值

我们可以对该模板添加更多有用的功能，这是根据条件过滤的元素。这是一个例子：

```
>>> even_squares = (x * x for x in range(10)
                     if x % 2 == 0)
```

这个生成器产出从零到九的所有偶数的平方。使用 `%`（模）运算符的过滤条件会过滤掉任何不能被二整除的值：

```
>>> for x in even_squares:
...     print(x)
0
4
16
36
64
```

让我们更新一下生成器表达式模板。通过添加 if 条件对元素进行过滤后，模板现在是这样的：

```
genexpr = (expression for item in collection
                  if condition)
```

该模式对应于一个相对简单但更长的生成器函数。最佳语法糖：

```
def generator():
    for item in collection:
        if condition:
            yield expression
```

一行的生成器表达式

因为生成器表达式是表达式，所以你可以将它们与其他语句一起使用。例如，你可以定义一个迭代器，并通过for循环立即使用它：

```
for x in ('Bom dia' for i in range(3)):
    print(x)
```

你还可以使用另一种语法技巧来使你的生成器表达式更简洁。如果将生成器表达式用作函数的单个参数，生成器表达式周围的括号可以去掉：

```
>>> sum((x * 2 for x in range(10)))
90
# Versus:
>>> sum(x * 2 for x in range(10))
90
```

这可以使你编写简洁而高效的代码。因为生成器表达式和基于类的迭代器以及生成器函数一样“实时”生成值，所以它们的内存使用效率非常高。

过犹不及.....

与列表推导类似，生成器表达式的复杂度比我们目前为止介绍的更高。通过嵌套for循环和链接过滤语句，它们可以涵盖更广泛的使用场景：

```
(expr for x in xs if cond1
      for y in ys if cond2
      ...
      for z in zs if condN)
```

上面的模式转换为下面的生成器函数：

```
for x in xs:
    if cond1:
        for y in ys:
            if cond2:
                ...
                for z in zs:
                    if condN:
                        yield expr
```

这是我要提醒的地方：

请不要像这样写深层嵌套的生成器表达式。从长远来看它们会很难维护。

这是一种“过犹不及”的情况，即使是一个简单漂亮的工具，过度使用也会导致难以阅读和难以调试的程序。

和列表解析一样，我个人尽量远离任何包含两层以上嵌套的生成器表达式。

生成器表达式是你工具箱中一个有用的工具，但这并不意味着应该将其用于你遇到的每个问题。对于复杂的迭代器，编写一个生成器函数，甚至是一个基于类的迭代器会更合适。

如果你需要使用嵌套生成器和复杂的过滤条件，通常最好抽出子生成器（这样你可以命名它们），然后将它们在顶层链接在一起。你会在下一章（连接迭代器）看到如何实现这种操作。

在你束手无策时，可以尝试不同的实现，然后选择可读性最高的实现。相信我，长远来看这可以节省你的时间。

重点

- 生成器表达式类似于列表解析。但是，它们不构造列表对象。相反，生成器表达式像基于类的迭代器或者生成器函数那样“实时”生成值。
- 生成器表达式一旦被消耗，就不能重新启动或重用。
- 生成器表达式最适合实现简单的“临时”迭代器。对于复杂的迭代器，最好编写生成器函数或基于类的迭代器。

6.7 连接迭代器

这是Python迭代器的另一个重要功能：通过连接多个迭代器，你可以编写高效的数据处理“管道”。我第一次大卫·比兹利（David Beazley）在PyCon的演讲中看到这种模式，那让我大吃一惊。

利用Python的生成器函数和生成器表达式，你可以迅速构建简洁而强大的迭代器链。在本章中，你将了解这种技术在实践中的表现以及如何在自己的程序中使用它。

快速回顾一下，生成器和生成器表达式是用Python编写迭代器的语法糖。他们精简了许多编写基于类的迭代器时需要的样板代码。

常规函数产生单个返回值时，生成器产出一系列值。你可以说他们在其生命周期产出了一个数据流。

例如，我可以定义以下生成器，通过保持一个运行中的计数器并在每次调用 `next()` 时产出一个新值，它可以产出从1到8的一系列整数值：

```
def integers():
    for i in range(1, 9):
        yield i
```

你可以通过在Python交互解释器上运行下面的代码进行确认：

```
>>> chain = integers()
>>> list(chain)
[1, 2, 3, 4, 5, 6, 7, 8]
```

到目前为止，还不是很有趣。但是，我们马上会对此进行更改。你会看到，可以将生成器彼此“连接”起来，以构建像管道一样工作的高效数据处理算法。你可以获取来自 `integers()` 生成器的数据“流”，然后将它们再次填入另一个生成器。例如，一个计算每个数的平方，然后将其传下去的生成器：

```
def squared(seq):
    for i in seq:
        yield i * i
```

这就是现在我们的“数据管道”或“生成器链”所要做的：

```
>>> chain = squared(integers())
>>> list(chain)
[1, 4, 9, 16, 25, 36, 49, 64]
```

我们可以继续向该管道添加新的代码基块。数据仅沿一个方向流动，通过明确定义的接口，每个处理步骤都和其他的步骤隔离起来。

这类似于Unix中管道的工作方式。我们将一系列的处理过程连接起来，以便每个过程的输出直接作为下一个的输入。

为什么不在我们的管道中添加另一个步骤来取每个值的负值，然后将其传递到链中的下一个处理步骤呢：

```
def negated(seq):
    for i in seq:
        yield -i
```

如果我们重建生成器链并在末尾添加 negated，那么输出如下：

```
>>> chain = negated(squared(integers()))
>>> list(chain)
[-1, -4, -9, -16, -25, -36, -49, -64]
```

关于生成器链，我最喜欢的事情是数据处理时一次只处理一个元素。生成器连中的处理步骤之间没有缓存：

1. integers 生成器产生单个值，比方说3。
2. 这将“激活” squared 生成器， squared 生成器将处理值，并将 ($3 \times 3 = 9$) 传递到下一级
3. squared 生成器产生的平方数立即进入 negated 生成器， negated 生成器将其修改为-9并再次产出它。

你可以继续扩展生成器链，以建立一个包含许多步骤的处理管道。它仍然可以高效执行，因为链中的每个步骤都是一个单独的生成器函数，修改也很轻松。

该处理管道中的每个单独的生成器函数都相当简洁。只需一点技巧，我们就可以用更少的代码定义这个流水线，并且不会牺牲很多可读性：

```
integers = range(8)
squared = (i * i for i in integers)
negated = (-i for i in squared)
```

请注意，我是如何用上一步输出产生的生成器表达式替换链中的每个处理步骤的。上面的代码等效于我们在本章中构建的生成器链：

```
>>> negated
<generator object <genexpr> at 0x1098bcb48>
>>> list(negated)
[0, -1, -4, -9, -16, -25, -36, -49]
```

使用生成器表达式的唯一缺点是它们不能接受参数，你无法在同一处理管道中多次重复使用相同的生成器表达式。

当然，在创建这些管道时，你可以混合使用生成器表达式和常规生成器。这会对提高复杂管道的可读性有帮助。

重点

- 生成器可以链接在一起以形成高效和可维护的数据处理管道。
- 链式生成器单独处理经过管道的每个元素。
- 生成器表达式可用于编写简洁的管道定义，但这会影响可读性。

source/python_tricks/7_字典的使用技巧.md

第七章 字典的使用技巧

7.1 字典的默认值

Python的字典的 `get()` 方法可以在查找字典的键对应的值时提供默认值。在许多情况下这会很方便。让我举一个简单的例子来说明这个特性。假设我们具有以下的数据结构，它将用户ID映射到用户名：

```
name_for_userid = {
    382: 'Alice',
    950: 'Bob',
    590: 'Dilbert',
}
```

现在，我们使用此数据结构编写一个 `greeting()` 函数，它会根据用户ID为用户返回问候语。我们的第一个实现可能是这样：

```
def greeting(userid):
    return 'Hi %s!' % name_for_userid[userid]
```

这是一个简单的字典查询。这个实现从技术上讲可以运行，但前提是用户ID在 `name_for_userid` 字典中存在。如果我们将无效的用户ID传递给 `greeting()` 函数会引发异常：

```
>>> greeting(382)
'Hi Alice!'
>>> greeting(33333333)
KeyError: 33333333
```

`KeyError` 异常并不是我们希望看到的结果。在找不到用户ID时，如果函数返回通用问候语会更好。

让我们实现这个想法。我们的第一种方法可能是简单地检查键是否在字典中，如果用户ID不存在，则返回默认的问候。

```
def greeting(userid):
    if userid in name_for_userid:
        return 'Hi %s!' % name_for_userid[userid]
    else:
        return 'Hi there!'
```

让我们看看这个 `greeting()` 的实现在之前的测试用例上表现如何：

```
>>> greeting(382)
'Hi Alice!'
>>> greeting(33333333)
'Hi there!'
```

好多了。现在，我们对未知用户我们可以有一个通用的问候，并且用户ID有效时，我们会保留个性化的问候。

但是这仍有改进的空间。尽管新的实现结果和我们预期的一样，并且看起来足够小巧，整洁，但是它仍然可以改进。我对当前方法的了解：

- 效率低下，因为它查询了两次字典。
- 冗长，比如，问候字符串的一部分有重复。
- 不Pythonic——官方Python文档推荐在这种场景下采用“寻求宽恕比许可容易”（EAFP）的编码风格：“这种通用的Python编码风格假定键或属性存在，并在假设错时捕获异常。”

一个更好的、遵循EAFP原则的实现可能会使用 `try...except` 块来捕获 `KeyError` 而不是显式地对key存在性进行检查：

```
def greeting(userid):
    try:
        return 'Hi %s!' % name_for_userid[userid]
    except KeyError:
        return 'Hi there'
```

就我们的最开始的需求而言，这个实现是没有问题的，现在，我们不再需要查询字典两次。但是我们仍然可以进一步改善它，并提供更整洁的方案。Python的字典有一个 `get()` 方法，该方法支持一个可用作后备值的“默认”参数：

```
def greeting(userid):
    return 'Hi %s!' % name_for_userid.get(userid, 'there')
```

调用 `get()` 时，它将检查字典中是否存在给定的键。如果存在，则返回键的值。如果不存在，则返回默认参数的值。`greeting` 函数的这种实现仍然可以按预期工作：

```
>>> greeting(950)
'Hi Bob!'
>>> greeting(333333)
'Hi there!'
```

我们最终的 `greeting()` 实现简明，整洁且仅使用Python标准库中的功能。因此，我相信这是这种特殊场景的最佳解决方案。

重点

- 在测试见是否在字典中时，避免显式使用 `key in dict` 进行检查。
- EAFP样式的异常处理或使用内置的 `get()` 方法是首选。
- 在某些情况下，标准库中的 `collections.defaultdict` 也很有用。

7.2 排序字典的乐趣和收益

Python字典没有固有的顺序。你可以对其进行遍历，但是不能保证迭代以任何特定顺序返回字典中的元素（尽管这种情况在Python 3.6已经发生变化）。

但是，获得字典的排序表示通常很有用的，排序可根据字典的键，值或其他衍生属性将键值对按任意顺序排列。假设你有一个字典`xs`，其内容如下：

```
>>> xs = {'a': 4, 'c': 2, 'b': 3, 'd': 1}
```

要获得此字典中键值对的排序列表，你可以先使用字典的 `items()` 方法，然后再对结果序列进行排序：

```
>>> sorted(xs.items())
[('a', 4), ('b', 3), ('c', 2), ('d', 1)]
```

为了比较序列，键值对元组使用Python标准的字典序进行排序。

为了比较两个元组，Python首先比较索引为零处的元素。如果它们不同，则索引为零处元素的比较结果就是两个元组比较的结果。如果它们相等，则比较索引为1的两个元素，依此类推。

现在，因为这些元组是从字典中获取的，每个元组中索引为零的元素都是字典的键，考虑到字典的键都是唯一的，所以元组索引为零处的元素都是唯一的。所以，这里没有特殊情况要处理。

在某些情况下，按字典顺序排序可能已经满足了你的需求。在其他情况下，你可能希望按值对字典进行排序。幸运的是，有一种方法可以完全控制元素的排序方式。你可以通过将 `key` 函数传递给 `sorted()` 函数来控制排序结果，`sorted()` 函数可以改变字典元素的比较方式。

`key` 函数只是在每个元素比较之前调用的普通Python函数。`key` 函数以字典元素作为输入，返回排序比较所需的“键”。

不幸的是，“`key`”一词在两种情况下都有使用，在这里 `key` 函数和字典键无关，它仅将每个输入元素映射为用于比较的值。

现在，也许我们应该看一个例子。相信我，在看到一些真实的代码后 `key` 函数会更容易理解。

假设你想获得按值排序的字典。要获得此结果，你可以使用下面的 `key` 函数，它通过查找元组中的第二个元素来返回每个键值对的值：

```
>>> sorted(xs.items(), key=lambda x: x[1])
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

看看存储键值对的结果列表是如何根据原始字典中存储的键值对的值进行排序的。值得花一些时间研究一下 key 函数的工作原理。你可以在各种Python上下文中应用它，这是一个很有用的概念。

实际上，这一概念非常普遍，以至于Python的标准库包含了 operator 模块。该模块实现了一些最常用的key函数作为即插即用的构建基块，例如，operator.itemgetter 和 operator.attrgetter。

这里有一个示例，说明如何在第一个示例中使用 operator.itemgetter 替换使用 lambda 的索引查找：

```
>>> import operator
>>> sorted(xs.items(), key=operator.itemgetter(1))
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

在某些情况下，使用 operator 模块可能会更清楚地传达你的意图。另一方面，使用简单的 lambda 表达式可能既可读又明确。在这种特殊情况下，我实际上更倾向于使用 lambda 表达式。

使用 lambda 作为自定义 key 函数的另一个好处是你可以以更精细的方式控制排序顺序。例如，你可以根据每个值的绝对数值对字典进行排序：

```
>>> sorted(xs.items(), key=lambda x: abs(x[1]))
```

如果你需要颠倒排序顺序让较大的值排在最前面，可以在调用 sorted() 时使用 reverse=True 关键字参数：

```
>>> sorted(xs.items(),
          key=lambda x: x[1],
          reverse=True)
[('a', 4), ('b', 3), ('c', 2), ('d', 1)]
```

就像我之前说过的，花点时间去掌握 key 函数在Python中的运行原理是完全值得的。他们为你提供了很多灵活性，通常可以使你免于编写代码进行数据结构转换的麻烦。

重点

- 创建字典和其他集合的排序的“视图”时，你可以使用 key 函数来控制排序结果。
- key 函数是Python中的重要概念。最常用的 key 函数甚至添加到了Python标准库的 operator 模块中。
- 函数是Python中的一等公民。这是一个在该语言中随处可见的强大功能。

7.3 用字典模拟switch/case语句

Python没有 switch/case 语句，因此有时需要写很长的 if...elif...else 语句来解决这个问题。在本章中，你会找到一个技巧，该技巧可以使用字典和一等函数模拟 switch/case 语句。听起来令人兴奋？太好了，我们开始！

想象一下，我们的程序中有以下if语句：

```
>>> if cond == 'cond_a':
...     handle_a()
... elif cond == 'cond_b':
...     handle_b()
... else:
...     handle_default()
```

当然，只有三种不同的条件，这并不太可怕。但是，试想一下，如果我们在此语句中有十个或更多个elif分支。事情就会有所不同。我认为过长的if语句是一种代码异味，它让程序更难阅读和维护。

处理长 if...elif...else 语句的一种方法是使用字典查找表进行替换，它可以模拟 switch/case 语句的行为。

这里的看法是利用Python具有一等函数这一事实。这意味着它们可以作为参数传递给其他函数，作为其他函数的值返回，赋值给变量和存储在数据结构中。

例如，我们可以定义一个函数，然后将其存储在列表中并在以后进行访问：

```
>>> def myfunc(a, b):
...     return a + b
...
>>> funcs = [myfunc]
>>> funcs[0]
<function myfunc at 0x107012230>
```

调用此函数的语法符合你的直觉期望，我们只需在列表中使用索引，然后使用“()”语法调用函数并将参数传递给它：

```
>>> funcs[0](2, 3)
5
```

现在，我们如何使用一等函数来减少很长的if语句的大小？这里的核心思想是定义字典，将输入条件的查找键映射到执行预期操作的函数上：

```
>>> func_dict = {
...     'cond_a': handle_a,
...     'cond_b': handle_b
... }
```

我们可以通过字典键查找来获取处理函数，然后调用它，而不是通过if语句检查每个条件然后进行过滤：

```
>>> cond = 'cond_a'
>>> func_dict[cond]()
```

这个实现已经可以工作了，至少当cond可以在字典中找到时是如此。如果cond不在字典中，我们将收到一个 KeyError 异常。

因此，让我们寻找一种方法来支持默认场景以与原来的 else 分支对应。幸运的是，所有Python字典都具有 get() 方法，它可以返回给定键值的值，如果键不存在则返回默认值。这正是我们在这里需要的：

```
>>> func_dict.get(cond, handle_default)()
```

最初，这段代码在语法上可能看起来很奇怪，但是当你对其进行分解后，会发现它的工作原理与前面的示例完全相同。我们使用Python的一等函数将 handle_default 传递给 get() 作为默认值。这样，在字典中找不到条件时，我们可以避免抛出 KeyError 异常，直接调用默认处理函数进行处理。

让我们看一个更完整的示例，该示例使用字典查找和一等函数替换 if 语句。通读以下示例后，你将能够看到将某些类型的 if 语句转换为基于字典的调度所需的模式。

我们将使用 if 语句编写另一个函数，然后对其进行转换。该函数采用字符串操作码，例如“add”或“mul”，然后对操作数x和y进行一些数学运算：

```
>>> def dispatch_if(operator, x, y):
...     if operator == 'add':
...         return x + y
...     elif operator == 'sub':
...         return x - y
...     elif operator == 'mul':
...         return x * y
...     elif operator == 'div':
...         return x / y
```

老实说，这只不过是另一个玩具示例（我不想用整页整页的代码让你觉得无聊），但可以很好地说明底层的设计模式。一旦“获得”模式，你将能够在各种不同的场景进行应用。

你可以通过在调用 dispatch_if() 函数时，传入字符串操作码和两个数字来尝试执行简单的计算：

```
>>> dispatch_if('mul', 2, 8)
16
```

```
>>> dispatch_if('unknown', 2, 8)
None
```

请注意，“未知”场景同样可以工作，因为Python在任何函数的末尾添加了一个隐式的 `return None` 语句。

到现在为止还挺好。让我们将原始的 `dispatch_if()` 转换为一个新函数，这个新函数使用字典将操作码映射到算术运算函数。

```
>>> def dispatch_dict(operator, x, y):
...     return {
...         'add': lambda: x + y,
...         'sub': lambda: x - y,
...         'mul': lambda: x * y,
...         'div': lambda: x / y,
...     }.get(operator, lambda: None)()
```

这种基于字典的实现与初始的 `dispatch_if()` 输出同样的结果。我们可以完全相同的方法调用这两个函数：

```
>>> dispatch_dict('mul', 2, 8)
16
>>> dispatch_dict('unknown', 2, 8)
None
```

如果是真正的“生产”代码，有一些方法可以这些代码进行改进。

首先，每次我们调用 `dispatch_dict()` 时，它都会创建一个临时字典和一堆用于操作码查找的 `lambda` 函数。这从性能角度来看并不理想。对于追求性能的代码，将字典创建为常量然后在调用函数时引用它会更有意义。我们不想在每次需要查找时都重新创建字典。

其次，如果我们真的想做一些诸如 `x + y` 的简单算术运算，那么最好使用Python的内置的 `operator` 模块而不是示例中使用的 `lambda` 函数。`operator` 模块为所有Python运算符提供了实现，例如`operator.mul`，`operator.div` 等。这是次要点。在此示例中，我有意使用了 `lambda` 使它更通用。这可以让你在其他场景下也可以应用该模式。

好了，现在你有了很多技巧，可以使用它们来简化一些 `if` 链，以防它们变得笨拙。请记住，这项技术并非在所有情况下都适用，有时你可以使用简单的 `if` 语句来获得更好的收益。

重点

- Python没有 `switch/case` 语句。但是在某些情况下，可以使用基于字典的调度表来避免过长的 `if` 链。
- Python的一等函数功能很强大。但是，强大的力量伴随着巨大的责任。

7.4 最疯狂的字典表达式

有时，你遇到的是一个真正有深度的小示例——如果你对代码进行了充分的思考，那么一行代码就可以教给你很多关于编程语言的知识。这样的代码片段感觉就像是Zenkōan：禅宗练习中使用的一个问题或陈述，引起了疑问并测试了学生的学习进度。

我们将在本节中讨论的小段代码就是这样的一个例子。乍一看，它看起来像是一个简单的字典表达式，但是如果更进一步思考，它将通过CPython解释器带你进入思维扩展之旅。

我从这个行代码中获得了巨大的帮助，有一次我将它打印在我的Python会议徽章上作为谈话的开始。这也使我与我的成员进行了一些有益的对话。

因此，事不宜迟，这里是代码段。花一点时间来思考一下下面的字典表达式及其计算结果：

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
```

我会在这里停一下…准备好了吗？这是在CPython解释器会话中上述字典表达式的结果：

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
{True: 'maybe'}
```

我承认，当我第一次看到这个结果时，我感到非常惊讶。但是，当你逐步调查发生了什么时，这一切都是合理的。因此，让我们考虑一下为什么我们会得到这个有点不直观的结果。

当Python处理字典表达式时，它首先构造一个新的空字典对象。然后按字典表达式中指定的顺序为其分配键和值。

因此，当我们对其进行分解后，上面的字典表达式等效于按顺序执行以下语句：

```
>>> xs = dict()
>>> xs[True] = 'yes'
>>> xs[1] = 'no'
>>> xs[1.0] = 'maybe'
```

奇怪的是，Python认为这个例子中使用的所有字典键都相等：

```
>>> True == 1 == 1.0
True
```

好的，先稍等片刻。我敢肯定，你可以接受`1.0 == 1`，但是为什么`True`也被认为等于`1`？我第一次看到这个表达式时，确实感到困惑。

在Python文档中进行了一些挖掘之后，我了解到Python将`bool`视为`int`的子类。在Python 2和Python 3中就是这种情况。

“布尔类型是整数类型的子类型，并且在几乎所有上下文中布尔值的行为类似于值0和1，唯一的例外是当转换为字符串时，会分别返回字符串'False'或'True'。

是的，这意味着你可以使用布尔作为Python中列表或元组的索引：

```
>>> ['no', 'yes'][True]
'yes'
```

为了保持清代码清晰，你不应该这样使用布尔值变量（为了你的同事能够保持理智）。

无论如何，让我们回到字典表达式中。

就Python而言，`True`，`1`和`1.0`都代表相同的字典键。在解释器执行字典表达式时，它会反复覆盖键`True`的值。这就解释了为什么最终的字典中只包含一个键。

在继续之前，让我们再看一下原始的字典表达式：

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
{True: 'maybe'}
```

为什么我们在这里仍然将`True`作为键？因为重复的覆盖，键不应该在最后变为`1.0`吗？

在CPython解释器源代码中进行了一些研究之后，我了解到Python的字典不会在与新对象相关联时更新键对象本身：

```
>>> ys = {1.0: 'no'}
>>> ys[True] = 'yes'
>>> ys
{1.0: 'yes'}
```

当然，这对性能优化是有意义的——如果键是相同的，那么为什么要花时间来更新原始的键呢？

在上一个示例中，你看到了初始`True`对象作为键从未被替换。因此，字典的字符串表示形式仍将键显示为`True`（而不是`1`或`1.0`）。

现在我们知道，字典中的值看上去只是因为相等而被覆盖。但是，事实证明，这种效果也不是仅由`__eq__`相等性检查引起的。

Python词典以哈希表数据结构作为底层实现。当我第一次看到这个令人惊讶的字典表达式时，我的直觉是这种行为与哈希冲突有关。

哈希表根据每个键的哈希值将其所有的键存储在不同的“桶”中。哈希值是从键中获取的，这个值是一个固定长度的数字，用来唯一标识键。

这允许快速查找。与将完整键对象与所有其他键进行比较并检查是否相等相比，在查找表中搜索键的数字哈希值要快得多。

但是，通常计算哈希值的方法并不完美。最终，实际上不同的两个或更多个键将具有相同的哈希值，并且最终将出现在相同的查找表存储桶中。

两个键具有相同的哈希值，称为哈希冲突，这是一种特殊情况，哈希表的插入和查找算法需要处理这种情况。

根据上面的分析，很可能哈希与我们从字典表达式中获得的令人惊讶的结果有关。因此，让我们在这里查看键的哈希值是否也在起作用。

我定义了下面的类作为我们的小侦探工具：

```
class AlwaysEquals:
    def __eq__(self, other):
        return True

    def __hash__(self):
        return id(self)
```

此类在两个方面很特殊。首先，由于其 `__eq__` 下划线方法始终返回 `True`，因此此类的所有实例都将假装它们等于任何其他对象：

```
>>> AlwaysEquals() == AlwaysEquals()
True
>>> AlwaysEquals() == 42
True
>>> AlwaysEquals() == 'waaat?'
True
```

其次，每个 `AlwaysEquals` 实例还将返回由内置 `id()` 函数生成的唯一哈希值：

```
>>> objects = [AlwaysEquals(),
    AlwaysEquals(),
    AlwaysEquals()]
>>> [hash(obj) for obj in objects]
[4574298968, 4574287912, 4574287072]
```

在CPython中，`id()` 返回内存中对象的地址，该地址被保证是唯一的。

通过此类，我们现在可以创建与其他任何对象都相同但具有唯一哈希值的对象。这样一来，我们可以测试字典的键是仅仅否因为他们的相等性比较而被覆盖。

而且，你看，即使它们始终相等，示例中的键也不会被覆盖：

```
>>> {AlwaysEquals(): 'yes', AlwaysEquals(): 'no'}
{<AlwaysEquals object at 0x110a3c588>: 'yes',
<AlwaysEquals object at 0x110a3cf98>: 'no' }
```

我们也可以将换个思路，看看是否哈希值相同就会导致键被覆盖：

```
class SameHash:
    def __hash__(self):
        return 1
```

`SameHash` 类的实例彼此不相等，但它们具有相同的哈希值1：

```
>>> a = SameHash()
>>> b = SameHash()
>>> a == b
False
>>> hash(a), hash(b)
(1, 1)
```

让我们看看当我们将 `SameHash` 类的实例用作字典的键时，Python的字典的行为：

```
>>> {a: 'a', b: 'b'}
{<SameHash instance at 0x7f7159020cb0>: 'a',
<SameHash instance at 0x7f7159020cf8>: 'b' }
```

从这个例子中可以看到，“键被覆盖”也不是仅由哈希值冲突引起的。

字典会检查两个值是否相等，并比较哈希值以确定两个键是否相同。我们尝试总结一下我们的调查结果：{True: 'yes', 1: 'no', 1.0: 'maybe'}字典表达式的计算结果为{True: 'maybe'}，因为键True，1和1.0的值都相等，并且它们具有相同的哈希值：

```
>>> True == 1 == 1.0
True
>>> (hash(True), hash(1), hash(1.0))
(1, 1, 1)
```

也许不再那么令人惊讶了，这就是字典的最终状态为什么是这样的原因：

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
{True: 'maybe'}
```

我们在这里涉及了很多主题，而这个Python技巧在开始时可能有点令人难以置信——这就是为什么我一开始将它与Zenkōan进行比较的原因。如果难以理解本节的内容，可以尝试在Python解释器中逐一运行这些代码示例。作为回报，你会掌握这些Python的底层知识。

重点

- 如果字典键相等（使用`__eq__`比较）且哈希值相同，则字典将它们视为相同的键。
- 意外的字典键冲突会导致令人惊讶的结果。

7.5 合并字典的多种方法

你是否曾经为一个Python程序构建过配置系统？这种系统的常见场景是采用具有默认配置选项的数据结构，然后允许从用户输入或其他配置源中有选择地覆盖默认值。

我经常发现自己使用字典作为表示配置信息的基础数据结构。因此，我经常需要一种方法来组合或合并配置默认值和用户配置值到单个字典中。

或概括地说：有时你需要一种将两个或多个字典合并为一个字典的方法，以便生成的字典包含所有源字典的键和值的合集。

在本章中，我将向你展示实现该目标的几种方法。我们首先来看一个简单的示例，以便进行一些讨论。

假设有以下两个源字典：

```
>>> xs = {'a': 1, 'b': 2}
>>> ys = {'b': 3, 'c': 4}
```

现在，你想要创建一个包含xs和ys的所有键和值的新字典zs。另外，如果你仔细阅读该示例，你会发现字符串“b”这个键在两个字典中都存在——我们还需要考虑重复键的冲突解决策略。

Python中“合并多个词典”的经典解决方案是使用字典的`update()`方法：

```
>>> zs = {}
>>> zs.update(xs)
>>> zs.update(ys)
```

如果你对其实现好奇，一个简单的`update()`实现可能像下面这样。我们只需遍历右侧字典的所有项，然后将每个键值对添加到左侧字典中，就可以覆盖现有的键：

```
def update(dict1, dict2):
    for key, value in dict2.items():
        dict1[key] = value
```

这会产生一个新的字典zs，这个字典现在包含在xs和ys中定义的所有键：

```
>>> zs
>>> {'c': 4, 'a': 1, 'b': 3}
```

我们调用`update()`的顺序决定了解决冲突的方式。最后一次更新影响了最终的结果，重复键“b”的值与第二个源词典ys的值3相等。

当然，只要你愿意，就可以扩展这个 `update()` 调用链，以便将任意数量的字典合并为一个。这是一种实用且易于理解的解决方案，可以在 Python 2 和 Python 3 中使用。

在 Python 2 和 Python 3 中另一种有效技术是组合使用内置的 `dict()` 与用来“解包”对象的**运算符：

```
>>> zs = dict(xs, **ys)
>>> zs
{'a': 1, 'c': 4, 'b': 3}
```

但是，就像重复进行 `update()` 调用一样，这个方法仅适用于合并两个字典，并且不能在一步中组合任意数量的字典。

从 Python 3.5 开始，** 运算符变得更加灵活。在 Python 3.5+ 中，还有另一种更漂亮的合并任意数量字典的方式：

```
>>> zs = {**xs, **ys}
```

该表达式的结果与 `update()` 调用链的结果完全相同。键和值以从左到右的顺序设置，因此这里的冲突解决策略是相同的：右侧优先，并且 `ys` 中的值将覆盖 `xs` 中相同键下的值。当我们查看合并操作产生的字典时，这一点就变得很清楚了：

```
>>> zs
>>> {'c': 4, 'a': 1, 'b': 3}
```

就我个人而言，我喜欢这种新语法的简洁性，同时它还有足够高的可读性。冗长和简洁之间总是要有一个很好的平衡，以使代码具有尽可能高的可读性和可维护性。

在这种场景下，如果我使用的是 Python 3，会更倾向于使用新语法。使用**运算符还比使用链接的 `update()` 要快，这是另一个好处。

重点

- 在 Python 3.5 及更高版本中，你可以使用**运算符在单个表达式中将多个字典合并为一个，遇到重复的键时，右边的字典的键会覆盖左边字典的键。
- 为了与旧版本的 Python 兼容，你可能需要使用内置的 `update()` 方法。

7.6 整洁地输出字典

你是否曾经尝试通过一堆的“`print`”语句来跟踪执行流程，以解决程序中的一个 bug？或者，也许你需要生成一条日志来打印一些配置设置……

某些 Python 中的数据结构作为文本字符串打印时，会很难阅读，这经常让我感到沮丧。例如，这是一个简单的字典。在解释器会话中打印时，键的顺序是任意的，并且结果字符串没有缩进：

```
>>> mapping = {'a': 23, 'b': 42, 'c': 0xc0ffee}
>>> str(mapping)
{'b': 42, 'c': 12648430, 'a': 23}
```

幸运的是，有一些易于使用的替代方法，可以直接将字典转换为字符串，从而获得更具可读性的结果。一种选择是使用 Python 内置的 `json` 模块。你可以使用 `json.dumps()` 以更好的格式打印 Python 字典：

```
>>> import json
>>> json.dumps(mapping, indent=4, sort_keys=True)
{
    "a": 23,
    "b": 42,
    "c": 12648430
}
```

这些设置会产生一个具有良好缩进的字符串表示形式，这个字符串表示还规范了键的顺序以提高可读性。

虽然看起来不错，而且可读性强，但这并不是一个完美的解决方案。使用 `json` 模块打印字典仅适用于包含基础类型的字典——尝试打印包含像函数这类非原始数据类型的字典会遇到麻烦：

```
>>> json.dumps({all: 'yup'})
TypeError: "keys must be a string"
```

使用 `json.dumps()` 的另一个缺点是，它无法将复杂的数据类型（例如集合）转换为字符串：

```
>>> mapping['d'] = {1, 2, 3}
>>> json.dumps(mapping)
TypeError: "set([1, 2, 3]) is not JSON serializable"
```

另外，你可能会遇到Unicode文本表示方式的麻烦——在某些情况下，你将 `json.dumps` 的输出复制粘贴到Python解释器会话中以重建原始字典对象。

内置的 `pprint` 模块是在Python中整洁打印对象的经典解决方案。这是一个例子：

```
>>> import pprint
>>> pprint pprint(mapping)
{'a': 23, 'b': 42, 'c': 12648430, 'd': set([1, 2, 3])}
```

你可以看到 `pprint` 能够打印数据集（例如集合），并且还可以以可重复的顺序打印字典键。与字典的标准字符串表示相比，这要养眼得多。

但是，与 `json.dumps()` 相比，它也无法在视觉上表示嵌套结构。根据情况，这可能是优点或缺点。由于可读性和格式的改进，我偶尔会使用 `json.dumps()` 打印字典，但前提是确定它们没有非原始数据类型。

重点

- Python中字典对象默认的字符串转换结果可能很难阅读。
- `pprint` 和 `json` 模块是Python标准库中内置的“高保真”选项。
- 同时使用 `json.dumps()` 和非基础类型的键和值时要小心，因为这会触发 `TypeError`

source/python_tricks/8_提高生产力的技巧.md

第八章 提高生产力的技巧

8.1 探索Python的模块和对象

你可以直接在Python解释器中以交互式的方式浏览模块和对象。这是一个被低估的功能，它很容易被忽视，尤其是当你从另一种语言切换到Python时。

许多编程语言在不查阅联机文档或不认真学习接口定义的情况下很难检查包或类。

Python是不同的——高效的开发人员会在交互式会话中花费大量时间与Python解释器进行交互。例如，为了弄清小段的代码和逻辑，我经常这样做，然后再将其复制到编辑器中正在处理的Python文件中。

在本节中，你会学到两种简单的技术，可用于在Python解释器中交互地探索Python的类和方法。

这些技术适用于任何Python——只需要在命令行使用 `python` 命令启动Python解释器，然后开始即可。在你无法访问高级编辑器或IDE上的系统调试会话时，这会非常有用，比如你正在通过网络在终端会话中进行工作。

准备好了？开始！假设你正在编写一个使用Python标准库中 `datetime` 模块的程序。你如何找到这个模块包含的函数或类，以及在这些类上有哪些方法和属性？

使用搜索引擎或在网络上查找Python的官方文档是实现这个目标的一种方法。但是，Python的内置 `dir()` 函数使你可以直接从Python的交互式会话中访问这些信息：

```
>>> import datetime
>>> dir(datetime)
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__',
 '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', '_divide_and_round',
 'date', 'datetime', 'datetime_CAPI', 'time',
 'timedelta', 'timezone', 'tzinfo']
```

在上面的示例中，我首先从标准库中导入了 `datetime` 模块，然后使用 `dir()` 函数对其进行了检查。在模块上调用 `dir()` 会提供按字母顺序排列的名称及属性列表。

因为Python中一切皆对象，所以相同的技术不仅适用于模块本身，而且适用于模块导出的类和数据结构。

实际上，你可以通过再次对感兴趣的单个对象调用 `dir()` 来继续深入探索。例如，以下是检查 `datetime.date` 类的方法：

```
>>> dir(datetime.date)
['__add__', '__class__', ..., 'day', 'fromordinal',
'isocalendar', 'isoformat', 'isoweekday', 'max',
'min', 'month', 'replace', 'resolution', 'strftime',
'timetuple', 'today', 'toordinal', 'weekday', 'year']
```

你看，`dir()` 提供了一个模块或类中可用内容的快速概述。如果你不记得特定类或函数的确切拼写，它可以在不中断你编码流程的情况下给予你恰当的帮助。

有时在对象上调用 `dir()` 会返回过多的信息——在复杂的模块或类上，你会得到很长的输出，这会让人难以快速进行阅读。你可以使用以下技巧来对属性列表进行过滤，只保留你感兴趣的属性：

```
>>> [__ for __ in dir(datetime) if 'date' in __.lower()]
['date', 'datetime', 'datetime_CAPI']
```

在这里，为了输出仅包含单词“date”的名称，我使用列表解析来过滤 `dir(datetime)` 的结果。注意我如何在每个名称上调用 `lower()` 方法，以确保过滤不区分大小写。

仅仅获取对象属性的原始列表并不足以解决当前的问题。那么，如何获得有关 `datetime` 模块的更多信息以及从中导出的函数和类的详细信息？

Python内置的 `help()` 函数可以助你一臂之力。借助它，你可以调用Python的交互式帮助系统来浏览任何对象的文档：

```
>>> help(datetime)
```

如果你在Python解释器会话中运行上述示例，则终端会显示 `datetime` 模块的文本帮助信息：

```
Help on module datetime:
NAME
    datetime - Fast implementation of the datetime type.
CLASSES
    builtins.object
        date
        datetime
        time
```

你可以使用上下光标键在文档中滚动。或者，你也可以敲空格键一次向下滚动几行。按q键可以退出交互帮助模式。这将带你返回解释器提示。不错的功能，对不对？

顺便说一句，你可以在任意Python对象（包括其他内置函数和你自己的Python类）上调用 `help()`。Python解释器会根据对象的属性及其文档字符串（如果有）自动生成这个文档。以下示例是 `help` 函数的所有有效用法：

```
>>> help(datetime.date)
>>> help(datetime.date.fromtimestamp)
>>> help(dir)
```

当然，`dir()` 和 `help()` 不会取代格式精美的HTML文档、搜索引擎或Stack Overflow搜索。但是，它们是无需离开Python解释器快速查找信息的好工具。它们还可以脱机使用，无需连接互联网即可工作，这在紧要关头非常有用。

重点

- 使用内置的 `dir()` 函数可以在解释器会话中交互式地探索Python模块和类。
- 内置的 `help()` 可让你直接从解释器中浏览文档（按q退出）。

8.2 使用Virtualenv隔离项目依赖

Python包含一个功能强大的打包系统，用于管理包依赖。你可能已经使用 `pip` 包管理命令安装了第三方软件包。

使用 `pip` 安装包的一个问题是它默认会将它们安装到全局的Python环境中。

当然，这可以使你安装的所有新软件包全局可用，这非常方便。但是，如果你要处理的多个项目需要使用同一软件包的不同版本，那么它也会很快变成一场噩梦。

例如，如果你的一个项目需要一个库的1.3版本，而另一个项目需要同一个库的1.4版本，该怎么办？

在全局安装软件包时，所有程序中只能有一个Python库版本。这意味着你将很快会遇到版本冲突，就像高地人^[^1]一样。^[^1]: 这个梗来自经典电视剧“Highlander: The Series”(《挑战者》)，因为苏格兰北部为高地(Highland)，而剧中的苏格兰裔男主角每次都只能战斗到剩下一个人

更糟的是，你可能还需要在不同的程序中使用不同版本的Python。例如，某些程序可能仍在Python 2上运行，而大多数新开发的程序都在Python 3上运行。或者，如果你的一个项目需要Python 3.3，而其他所有项目都在Python 3.6上运行，该怎么办？

除此之外，在全局安装Python软件包还可能带来安全风险。修改全局环境通常需要你使用超级用户（root/admin）运行 `pip install` 命令。因为在安装新软件包时pip会从Internet下载并执行代码，所以通常不建议这样做。希望代码是值得信赖的，但是谁知道它真正会做什么呢？

使用虚拟环境

解决这些问题的方法是用虚拟环境将你的Python环境进行隔离。它们使你可以按项目分隔Python依赖项，并使你能够选择不同版本的Python解释器。

虚拟环境是一个隔离的Python环境。从物理上讲，它位于一个文件夹内，这个文件夹包含Python项目所需的所有包和其他依赖项，例如代码库和解释器运行时。（在底层，为了节约存储空间，这些文件可能不是真实的副本，而只是链接。）

为了演示虚拟环境的工作原理，我将向你快速介绍如何设置一个新环境（简称为 `virtualenv`），然后在其中安装第三方程序包。

首先，让我们检查一下全局Python环境所在的位置。在Linux或macOS上，我们可以使用`which`命令行工具来查找 `pip` 的路径：

```
$ which pip3
/usr/local/bin/pip3
```

为了保持美观和独立，我通常将虚拟环境放在项目文件夹中。但是，你也可以在某个地方有一个专用的python环境目录，用来容纳跨项目的所有环境。这是你的选择。

让我们创建一个新的Python虚拟环境：

```
$ python3 -m venv ./venv
```

这需要一点时间，它会在当前目录中创建一个`venv`文件夹，文件夹中会包含一个基准的Python 3环境：

```
$ ls venv/
bin include lib pyvenv.cfg
```

如果你检查`pip`的活动版本（使用 `which` 命令），你会看到它仍指向全局环境，在我的环境下是 `/usr/local/bin/pip3`：

```
(venv) $ which pip3
/usr/local/bin/pip3
```

这意味着，如果你现在安装软件包，它们仍然会在全局Python环境中运行。创建虚拟环境文件夹是不够的，你需要显式激活新的虚拟环境，以便以后运行`pip`命令时引用该文件夹：

```
$ source ./venv/bin/activate
(venv) $
```

运行 `activate` 命令会将当前的Shell会话配置为使用虚拟环境中的Python和 `pip` 命令。

请注意，这会更改你的Shell提示符，在括号内会包含活动的虚拟环境的名称：(venv)。让我们检查一下哪个 `pip` 现在处于活动状态：

```
(venv) $ which pip3
/Users/dan/my-project/venv/bin/pip3
```

你看，执行`pip3`命令现在会运行在虚拟环境中的 `pip` 而不是全局环境中的。Python解释器也是如此。现在，从命令行运行`python`也会从`venv`文件夹中加载解释器：

```
(venv) $ which python
/Users/dan/my-project/venv/bin/python
```

请注意，这仍然是一片空白，一个完全干净的Python环境。执行 `pip list` 将显示几乎没有已安装的包，仅仅包括支持 `pip` 本身所需的基准模块：

```
(venv) $ pip list
pip (9.0.1)
setuptools (28.8.0)
```

让我们继续，现在将Python软件包安装到虚拟环境中。你想使用熟悉的 `pip install` 命令：

```
(venv) $ pip install schedule
Collecting schedule
  Downloading schedule-0.4.2-py2.py3-none-any.whl
    Installing collected packages: schedule
      Successfully installed schedule-0.4.2
```

你会注意到两个重要的变化。首先，你将不再需要管理员权限才能执行此命令。其次，使用激活的虚拟环境安装或更新包意味着所有文件都将最终存储在虚拟环境目录的子文件夹中。

因此，你的项目依赖项将与系统上的所有其他Python环境（包括全局环境）在物理上隔离。实际上，你会为每个项目获得一个单独的Python运行时的克隆。

通过再次运行 `pip list`，你可以看到 `schedule` 库已成功安装到新环境中：

```
(venv) $ pip list
pip (9.0.1)
schedule (0.4.2)
setuptools (28.8.0)
```

如果我们使用`python`命令启动一个Python解释器会话，或者使用它运行一个独立的`.py`文件，它将使用安装在虚拟环境中的Python解释器和依赖项——只要该环境仍处于激活状态。

但是，如何再次停用或“离开”虚拟环境呢？与 `activate` 命令类似，有一个 `deactivate` 命令可以让你回到全局环境：

```
(venv) $ deactivate
$ which pip3
/usr/local/bin
```

使用虚拟环境将有助于使你的系统整洁，并使Python依赖项井井有条。最佳做法是，所有Python项目均应使用虚拟环境来隔离依赖项并避免版本冲突。

了解和使用虚拟环境还可以使你走上正确的道路，你可以使用更高级的依赖关系管理方法，例如使用`requirements.txt`文件指定项目依赖。

如果你想深入了解这个主题以获得其他提高效率的提示，请务必看看我的[课程](#)。

重点

- 虚拟环境可以使项目依赖相互隔离。它们帮助你避免Python包与运行时的版本冲突。
- 最佳做法是，所有Python项目均使用虚拟环境来存储其依赖项。这将有助于避免一些头痛的问题。

8.3 字节码

当CPython解释器执行程序时，它首先将其翻译为一系列字节码指令。字节码是Python虚拟机的一种中间语言，用来进行性能优化。

解释器不是直接执行人类可读的源代码，而是使用紧凑的数字代码，常量和引用来表示编译器解析和语义分析的结果。

这样可以使重复执行的程序或一部分程序速度更快，并且更加节省内存。编译步骤产生的字节码被缓存磁盘上的`.pyc`和`.pyo`文件中，以便第二次执行相同的Python文件时会更快。

所有这些对程序员都是完全透明的。你不必知道会有这个中间翻译步骤，也不必知道Python虚拟机如何处理字节码。实际上，字节码格式被视为实现细节，不能保证在不同Python版本之间保持稳定或兼容。

但是，我发现观察具体的过程和CPython解释器背后提供的抽象很有启发性。至少了解一些底层原理可以帮助你编写出性能更高的代码（这很重要）。而且也很有趣。

让我们用这个简单的`greet()`函数作为示例，我们可以使用它来理解Python的字节码：

```
def greet(name):
    return 'Hello, ' + name + '!'

>>> greet('Guido')
'Hello, Guido!'
```

还记得我说过CPython在运行之前先将我们的源代码翻译成中间语言吗？好吧，如果是这样，我们应该能够看到这个编译步骤的结果。我们确实可以。

每个函数都有一个`__code__`属性（在Python 3中），我们可以用来获取虚拟机的指令，常量和由`greet`函数使用的变量：

```
>>> greet.__code__.co_code
b'd\x01|\x00\x17\x00d\x02\x17\x00S\x00'
>>> greet.__code__.co_consts
(None, 'Hello, ', '!')
>>> greet.__code__.co_varnames
('name',)
```

你可以看到 `co_consts` 包含了我们的问候字符串的一部分。常量和代码分开保存以节省存储空间。常量意味着它们永远不能修改，并且可以在多个地方互换使用。

因此，Python 无需在 `co_code` 指令流中重复实际的常量值，而是将常量分别存储在查找表中。然后，指令流可以使用索引引用在查找表中的常量。存储在 `co_varnames` 字段中的变量也是如此。

我希望这个概念可以更加清晰。但是看着 `co_code` 指令流仍然让我感到有些不安。显然，这种中间语言对 Python 虚拟机来说更容易使用，对于人来说则不然。毕竟，那是基于文本的源代码的用途。

使用 CPython 的开发人员也意识到了这一点。因此，他们给了我们另一个称为反汇编程序的工具，使检查字节码更加容易。

Python 的字节码反汇编程序位于标准库中的 `dis` 模块中。因此，我们可以将其导入并在 `greet` 函数上调用 `dis.dis()`，这样就可以获得其字节码更易读的表示形式：

```
>>> import dis
>>> dis.dis(greet)
2 0 LOAD_CONST 1 ('Hello, ')
2 LOAD_FAST 0 (name)
4 BINARY_ADD
6 LOAD_CONST 2 ('!')
8 BINARY_ADD
10 RETURN_VALUE
```

反汇编的主要工作是拆分指令流，并为其中的每个操作码起一个人类易读的名称，例如 `LOAD_CONST`。你还可以看到常量和变量引用现在如何与字节码交织并完整打印，可以给我们免去在查找表查找 `co_const` 或 `co_varnames` 的麻烦。整洁！

通过查看操作码，我们可以理解 CPython 表示和执行原始 `greet()` 函数中 `'Hello' + name + '!' '语句的原理。`

它首先检索索引 1 处的常量（`'Hello'`），将其放入栈中。然后，它加载变量 `name` 的内容，并将它们放入栈中。

栈是虚拟机内部工作时使用的数据结构。虚拟机有不同的类别，其中一类称为堆栈机。CPython 的虚拟机就是这种堆栈机的实现。如果整个虚拟机都以栈命名，那么你可以想象栈这种数据结构起着怎样核心的作用。

顺便说一句，这只是触摸到了皮毛。如果你对这个主题感兴趣，可以在本章最后找到推荐书籍。阅读有关虚拟机理论的内容很有启发性（并且有很多乐趣）。

栈作为抽象数据结构的有趣之处在于，它只支持两种操作：入栈和出栈。入栈将一个值添加到栈的顶部，出栈则删除并返回最上面的值。与数组不同，栈无法访问顶部元素“下方”的元素。

如此简单的数据结构具有如此多的用途，这让我感到非常着迷。我又被迷住了...

假设栈开始是空的。在执行了前两个操作码之后，虚拟机栈的内容如下（0 是最上面的元素）：

```
0: 'Guido' (contents of "name")
1: 'Hello, '
```

`BINARY_ADD` 指令从栈中弹出两个字符串值，将它们连接起来，然后将结果再次压入栈：

```
0: 'Hello, Guido'
```

然后使用 `LOAD_CONST` 将感叹号字符串压入栈：

```
0: '!'
1: 'Hello, Guido'
```

下一个 `BINARY_ADD` 操作码再次将两者结合在一起以生成最终的问候字符串：

```
0: 'Hello, Guido!'
```

最后一个字节码指令是 RETURN_VALUE，它告诉虚拟机当前栈顶的内容是该函数的返回值，因此可以将其传递给调用方。

我们刚刚追溯了CPython虚拟机如何在内部执行 greet() 函数。那难道不是很酷吗？关于虚拟机，还有更多可以讲解的内容，但这不是这本书的主题。如果这引起了你的兴趣，我强烈建议你对这个有趣的话题进行更多阅读。

定义自己的字节码语言并创建小型的虚拟机实验会很有趣。我推荐的一本这个主题的书：Wilhelm和Seidl撰写的*Compiler Design: Virtual Machines*。

重点

- CPython首先将程序转换为中间字节码，然后在基于栈的虚拟机上运行字节码来执行程序。
- 你可以使用内置的 dis 模块查看运行情况并检查字节码。
- 在虚拟机这个主题上学习是值得的。

source/python_tricks/9_总结.md

第九章 总结

恭喜，您一直坚持到了最后！是时候给自己一点表扬了，因为大多数人在买了一本书后，甚至从不拆开它，还有些人根本不会读到第一章以后的部分。

但是，既然您已经阅读了这本书，那么这就是真正开始的地方——阅读和做事之间有很大的区别。带着您在这本书中学到的新技能和技巧，然后进行使用。不要让这本书只是您阅读的另一本编程书籍。

从现在开始，如果您开始在代码中添加一些Python的高级功能，会发生什么呢？在这里使用一个整洁的生成器表达式，在那里优雅地使用了with语句……

如果做得正确，您将立即获得同僚的关注——通过一种好的方式。通过一些练习，您将毫无困难地应用这些高级Python功能，并且会仅在有意义的地方使用它们，此外还有助于使您的代码更具表现力。

相信我，您的同事将在一段时间后注意到这些。如果他们问您问题，请大方一点去帮助别人。帮助周围的人了解您所知道的。您甚至可以在接下来的几周内为您的同事提供一些有关“编写干净Python代码”的演讲。本书中的示例可以随意使用。

作为Python程序员，做一项出色的工作和被看到做一项出色的工作是有区别的。不要害怕冒出来。如果您与周围的人分享您的技能和新的知识，您的职业生涯将大为受益。

我在自己的职业和项目中遵循相同的思维方式。因此，我一直在寻找改善本书和其他Python培训资料的方法。如果您想让我知道一个错误，或者您有任何疑问或想提供一些建设性的反馈，请给我发电子邮件mail@dbader.org。

Python编程快乐！

— Dan Bader P.S：你可以通过网络访问我的[网站](#)和[YouTube](#)，并继续你的Python之旅。

9.1 面向Python开发人员的免费每周提示

您是否在寻找每周的Python开发技巧，以提高生产力并简化工作流程？好消息是我正在运行一个免费的电子邮件简报，为像您一样的Python开发人员提供服务。

我发送的电子邮件不是典型的“热门文章列表”。相反，我的目标是每周以短论文的形式分享至少一个原创思想。

如果您想了解相关信息，请访问dbader.org/newsletter，然后在注册表单中输入您的电子邮件地址。我期待着与您见面！