[笔记][Learning Python][3. 语句和语法]

Python

---

10. Python 语句简介

10.1 重温 Python 的知识结构

1. 程序由模块构成
2. 模块包含语句
3. 语句包含表达式
4. 表达式创建并处理对象

> 程序、包、模块、类、函数、语句、表达式

10.2 Python 的语句

Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	`a, b = 'good', 'bad'`
Calls and other expressions	Running functions	`log.write("spam, ham")`
`print` calls	Printing objects	`print('The Killer', joke)`
`if/elif/else`	Selecting actions	`if "python" in text:` `  print(text)`
`for/else`	Iteration	`for x in mylist:` `  print(x)`
`while/else`	General loops	`while X > Y:` `  print('hello')`
`pass`	Empty placeholder	`while True:` `  pass`
`break`	Loop exit	`while True:` `  if exittest(): break`
`continue`	Loop continue	`while True:` `  if skiptest(): continue`
`def`	Functions and methods	`def f(a, b, c=1, *d):` `  print(a+b+c+d[0])`
`return`	Functions results	`def f(a, b, c=1, *d):` `  return a+b+c+d[0]`
`yield`	Generator functions	`def gen(n):` `  for i in n: yield i*2`

`global`	Namespaces	`x = 'old'`
		`def function():`
		`  global x, y; x = 'new'`

Statement	Role	Example
		def function(): nonlocal x; x = 'new'
import	Module access	import sys
from	Attribute access	from sys import stdin

class	Building objects	class Subclass(Superclass):
		  staticData = []
		  def method(self): pass

try/except/ finally	Catching exceptions	try:
		  action()
		except:
		  print('action error')

raise	Triggering exceptions	raise EndSearch(location)
assert	Debugging checks	assert X > Y, 'X too small'
with/as	Context managers (3.X, 2.6+)	with open('data') as myfile: process(myfile)
del	Deleting references	del data[k] del data[i:j] del obj.attr del variable

> 补充：`assert` 用法

```
>>> X = 1
>>> Y = 2
>>> assert X > Y, 'X too small'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: X too small
```

- `print` 在 `3.X` 不是保留字，也不是语句，而是一个内置函数。
- 从 `2.5` 开始，`yield` 不是语句，它是表达式，同时还是保留字。
- `2.X` 中 `nonlocal` 不可用。
- `3.X` 中 `exec` 是一个函数，而 `2.X` 中他是一条语句。
  但是 `2.X` 支持带有圆括号的形式，所以在 `2.X` 中可以通用地写 `3.X` 的调用形式。
- `2.5` 合并了 `try/except` 和 `try/finally`
- `2.5` 的上下文管理器 `with ... as ...` 默认不可用，要 `from __future__ import with_statement` 开启

两种不同的 `if`
类 `C` 语言的语法

```
if (x > y) {
    x = 1;
    y = 2;
```

```
}
```

`Python` 的写法：

```
if x > y:
    x = 1
    y = 2
```

所有 `Python` *复合语句*（内嵌了其他语句的语句）都有相同的一般形式，首行以冒号 `colon character` 结尾，之后的嵌套代码块要缩进。

```
Header line:
    Nested statement block
```

`Python` 添加了：

- 冒号

`Python` 减少了：

- 小括号是不必要的
- 行终止就是语句终止，不要分号
- 缩进的结束就是代码块的结束，不要大括号

`C++` 的循环头

```
while (x > 0) {
   --------;
}
```

`C` 语言的语句：

```
if (x)
    if (y)
        statement1;
else
    statement2;
```

`else` 是属于 `if(y)` 而不是 `if(x)`
而在 `Python` 里面，垂直对齐的 `if` 就是逻辑上的 `if`，外层的 `if(x)`。
`Python` 是一种 `WYSIWYG` 所见即所得的语言。

不能混合使用制表符和空格缩进，`3.X` 会报错。

几种特殊情况

- `;` 可以作为语句分隔符，把多条语句写在一行中
  如果放到一起的语句本身是一条复合语句，不用这么做
- 包括在括号里的代码可以横跨多行（小中大三种括号都可以）
  分行书写的缩进是任意的，但是为了可读性，最好是对齐

```
if (A == 1 and
    B == 2 and
    C == 3):
        print('spam' * 3)
```

- 使用反斜杠续行符也可以换行，但是不推荐使用
  在反斜杠后面加 *任何字符* 都会报错
- 单行的复合语句：`if x > y: print(x)`
  类似的还有单行 `while`、`for` 循环
  复合语句体也可以用分号隔开写在一行，但是不受欢迎
  中断循环的单行 `if` 语句加 `break` 的情况比较常见。

10.3 简短示例：交互式循环

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(reply.upper())
```

`2.X` 兼容的代码

```
import sys
if sys.version[0] == '2': input = raw_input # 2.X compatible
```

其中 `sys.version` 可以这么查看：

```
>>> import sys
>>> sys.version
'3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD
64)]'
>>> sys.version[0]
'3'
```

当一个异常发生时，我们会直接对它进行响应，而不是预先检测一个错误。

```
while True:
    reply = input('Enter text:')
```

```
        if reply == 'stop': break
        try:
            print(int(reply) ** 2)
        except:
            print('Bad!' * 8)
    print('Bye')
```

10.4 本章小结

10.5 本章习题

5. 如何在一行上编写复合语句？
   复合语句的主体可以移到开头行的冒号后面，但前提是主体仅由非复合语句组成。
6. `Python` 中是否有正当的理由在语句的末尾使用分号呢？
   只有当你需要把多条语句挤进一行代码时。
   即使在这种情况下，也只有当所有语句都是非复合语句时才行得通。

11. 赋值、表达式和打印

11.1 赋值语句

- 赋值语句创建对象引用
- 变量在首次赋值时会被创建
  一旦赋值后，每当这个变量出现在表达式中时，就会替换成其引用的值。
- 变量在引用前必须先赋值
- 某些操作会隐式地进行赋值
  比如模块导入、函数和类的定义、`for` 循环变量以及函数参数，都是隐式赋值运算

Table 11-1. Assignment statement forms

Operation	Interpretation
`spam = 'Spam'`	Basic form
`spam, ham = 'yum', 'YUM'`	Tuple assignment (positional)
`[spam, ham] = ['yum', 'YUM']`	List assignment (positional)
`a, b, c, d = 'spam'`	Sequence assignment, generalized
`a, *b = 'spam'`	Extended sequence unpacking (Python 3.X)
`spam = ham = 'lunch'`	Multiple-target assignment
`spams += 42`	Augmented assignment (equivalent to `spams = spams + 42`)

元组及列表解包赋值
第二行和第三行
第二行：`Python` 自动在右侧新建了一个元组

序列赋值
任何值的序列都可以赋值给任何名称的序列

扩展的序列解包
第五行
提供了手动切片的一个简单的替代方案

多目标赋值
并没有产生独立副本，而是得到了同一个对象。

> 链式赋值。因为 `Python` 的赋值表达式是有值的。

增量赋值
在 `Python` 中，每一种二元表达式运算符都有对应的增量赋值语句。

序列赋值

```
% python
>>> nudge = 1 # Basic assignment
>>> wink = 2
>>> A, B = nudge, wink # Tuple assignment
>>> A, B # Like A = nudge; B = wink
(1, 2)
>>> [C, D] = [nudge, wink] # List assignment
>>> C, D
(1, 2)
```

交换变量时，`Python` 会在右侧创建一个临时元组
`Python` 中原本的元组和列表赋值形式已经得到了推广，从而能在右侧接收任意类型的序列（实际

上，也可以是可迭代对象），只要长度等于左侧序列即可。

我们甚至可以复制内嵌的序列，只要左侧对象的序列嵌套的形状必须与右侧对象的形状相同。

> 【注】你有一个鼻子，我也有一个鼻子。

```
>>> ((a, b), c) = ('SP', 'AM') # Paired by shape and position
>>> a, b, c
('S', 'P', 'AM')
```

还可以用于 `for` 循环：

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ... # Simple tuple assignment
for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: ... # Nested tuple assignm
ent
```

还可以将一系列整数赋值给一组变量，相当于其他语言中的枚举数据类型：

```
>>> red, green, blue = range(3)
>>> red, blue
(0, 2)
```

还有一个元组赋值语句的应用场景，在循环中把序列分割为开头和剩余两部分：

```
>>> L = [1, 2, 3, 4]
>>> while L:
... front, L = L[0], L[1:] # See next section for 3.X * alternative
... print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

`3.X` 中的扩展序列解包

`*` 星号后面的变量会收集一个 列表。

```
>>> seq = [1, 2, 3, 4]
>>> a, *b = seq
>>> a
1
>>> b
[2, 3, 4]
```

```
>>> print(type(b))
<class 'list'>
```

> 【注】我以前都一直误以为是一个元组！

无论带星号的名称出现在哪里，这个名称都会被赋值一个列表，而这个列表会收集在该位置上的所有待分配对象。

扩展的序列解包语法对于任何 *可迭代对象* 都有效。

```
>>> a, *b = 'spam'
>>> a, b
('s', ['p', 'a', 'm'])

>>> a, *b, c = 'spam'
>>> a, b, c
('s', ['p', 'a'], 'm')

>>> a, *b, c = range(4)
>>> a, b, c
(0, [1, 2], 3)
```

切片操作得到的对象类型与原类型一致，而解包语法总是得到一个列表。

更加简洁的取头元素的写法：

```
>>> L = [1, 2, 3, 4]
>>> while L:
... front, *L = L # Get first, rest without slicing
... print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

边界情况：匹配到单个元素，会得到一个单元素的列表；没有剩余的元素，则会得到一个空列表，不会报错。

```
>>> seq = [1, 2, 3, 4]
>>> a, b, c, *d = seq
>>> print(a, b, c, d)
1 2 3 [4]

>>> a, b, c, d, *e = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
```

```
>>> a, b, *e, c, d = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
```

报错的情况：以下三种情况都会报错

- 使用了多个带星号的名称
- 名称数目小于序列长度，同时没有星号名称
- 带星号的名称没有在一个列表或元组中

```
>>> a, *b, c, *d = seq
SyntaxError: two starred expressions in assignment

>>> a, b = seq
ValueError: too many values to unpack (expected 2)

>>> *a = seq
SyntaxError: starred assignment target must be in a list or tuple

>>> *a, = seq
>>> a
[1, 2, 3, 4]
```

在 `2.X` 只能只能使用索引和切片来实现解包赋值。
两种方法都可以满足 “第一项，剩余项” 的编程需求。

```
>>> seq
[1, 2, 3, 4]

>>> a, *b = seq # First, rest
>>> a, b
(1, [2, 3, 4])

>>> a, b = seq[0], seq[1:] # First, rest: traditional
>>> a, b
(1, [2, 3, 4])
```

解包赋值也可以用于 `for` 循环

```
for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
    ...
```

> 意思是 `2.X` 不支持星号解包

多目标赋值，只有一个对象，多个变量共享引用。
`a = b = c = 'spam'`
对于不可变类型而言没有什么问题。

```
>>> a = b = 0
>>> b = b + 1
>>> a, b
(0, 1)
```

但是如果是可变类型，就会共享引用：

```
>>> a = b = []
>>> b.append(42)
>>> a, b
([42], [42])
```

> 发生联动修改

两种解决方案：

- 使用单独的赋值语句
- 使用元组赋值

```
# 单独赋值
>>> a = []
>>> b = [] # a and b do not share the same object
>>> b.append(42)
>>> a, b
([], [42])

# 元组赋值
>>> a, b = [], [] # a and b do not share the same object
```

增量赋值
`augmented assignment`
借鉴了 `C` 语言。

Table 11-2. Augmented assignment statements

X += Y	X &= Y	X -= Y	X |= Y
X *= Y	X ^= Y	X /= Y	X >>= Y
X %= Y	X <<= Y	X **= Y	X //= Y

增量赋值语句的优点：

- 减少程序员的输入
- 左侧只需计算一次
  `X = X + Y` 中，`X` 要算两次
  而 `X += Y` 中，`X` 只要算一次
  所以增量赋值通常执行地更快
- 增量赋值对于可以原位置修改的对象，不会复制，而是自动选择在原位置修改
  对于增量赋值，原位置运算能作为可变对象的一种优化。

列表在尾部追加元素：可以用拼接或者 `append`。

```
>>> L = [1, 2]
>>> L = L + [3] # Concatenate: slower
>>> L
[1, 2, 3]
>>> L.append(4) # Faster, but in place
>>> L
[1, 2, 3, 4]
```

列表在尾部追加一组元素：可以用拼接或者 `extend`。
（当然还可以使用 切片赋值 操作：`L[len(L):] = [11, 12, 13]`）

```
>>> L = L + [5, 6] # Concatenate: slower
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend([7, 8]) # Faster, but in place
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

这两种情况，拼接更不容易被共享对象引用的副作用影响，但是通常会比等价的原位置形式运行的更慢。拼接操作必然会创建一个新对象，把加号左侧和右侧的列表都复制到其中。相反，原位置方法调用直接在一个内存块末尾添加项。（内部实现其实更复杂些）

使用增量赋值的时候，会自动调用较快的 `extend` 方法，而不是较慢的 `+` 运算。

列表的 `+=` 并不总是等于 `+` 和 `=` 拆开来写。
对于列表，`+=` 接收任意的序列（就像 `extend`），但是拼接一般情况下不接受。

```
>>> L = []
>>> L += 'spam' # += and extend allow any sequence, but + does not!

>>> L
['s', 'p', 'a', 'm']
>>> L = L + 'spam'
TypeError: can only concatenate list (not "str") to list
```

> `L.extend(可迭代对象)` 其实接收可迭代对象！这个以前我真不知道！

注意增量赋值是在原位置修改，如果有共享引用的变量，会导致联动修改。

```
>>> L = [1, 2]
>>> M = L # L and M reference the same object
>>> L = L + [3, 4] # Concatenation makes a new object
>>> L, M # Changes L but not M
([1, 2, 3, 4], [1, 2])

>>> L = [1, 2]
>>> M = L
>>> L += [3, 4] # But += really means extend
>>> L, M # M sees the in-place change too!
([1, 2, 3, 4], [1, 2, 3, 4])
```

变量名命名规则

- 语法：下划线或字母 + 任意数目的字母、数字或下划线
- 区分大小写
- 禁止使用保留字

Table 11-3. Python 3.X reserved words

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

`2.X` 中保留字有所不同：

- `print` 是语句，因此也是保留字 `reserved word`
- `exec` 是保留字，因为它是语句
- `nonlocal` 不是保留字，在 `2.X` 中不存在这条语句

更早的 `Python` 有所不同：

- `with` 和 `as` 在 `Python 2.6` 之前不是保留字，只有开启上下文管理器之后才是
- `yield` 在 `Python 2.3` 之前不是保留字，只有在生成器函数可用后才是
- `yield` 从 `2.5` 开始从语句变为表达式，但它仍然是一个保留字，而不是一个内置函数名

无法通过赋值来重新定义保留字，比如 `and = 1` 会报错。（至少在 `CPython` 是这样）

表中前三项比较特殊，大小写混合，而且 `2.X` 中可以赋值为其他对象。但在 `3.X` 它们彻底成为了保留字。

因为 `import` 语句中的模块名会成为脚本中的变量，变量命名规则也会限制模块的文件名。
可以编写 `and.py` 和 `my-code.py` 作为顶层脚本运行，但是不能导入它们。

`Python` 中的废弃协议

- 先让新功能成为可选的
- 对旧功能发布废弃 `deprecation` 警告
- 最终启用新功能

命名惯例

- 以单一下划线开头的名称 `_X` 不会被 `from module import *` 语句导入。
- 前后双下划线的名称是系统定义的名称 `__X__`，对接时期有特殊意义。
- 以双下划线开头，但结尾没有双下划线的名称 `__X` 会被重整（`31` 章的伪私有属性 `psuedoprivate attribute`）。

`built-ins` 内置名称，预先定义但是不是保留字，所以一般不会 `open = 42`。

名称没有类型，但是对象有类型
名称（或者叫变量）永远只是对象的引用。
名称存在于作用域中，而作用域决定了名称可以在何处使用。
一个名称赋值的位置，决定了它在哪里可见。

11.2 表达式语句

表达式可以作为语句，也就是让表达式独占一行。
表达式通常在以下两种情况下用作语句：

- 调用函数和方法：一些函数和方法在进行工时不返回值。这样的函数有时在其他编程语言中被称为过程 `procedure`。
- 在交互式命令行下打印值：交互模式会回显表达式的结果，可以看作是 `print` 的简写。

Table 11-4. Common Python expression statements

Operation	Interpretation
`spam(eggs, ham)`	Function calls
`spam.ham(eggs)`	Method calls
`spam`	Printing variables in the interactive interpreter
`print(a, b, c, sep='')`	Printing operations in Python 3.X
`yield x ** 2`	Yielding expression statements

表达式可以用作语句（就是这里的表达式语句 `expression statement`）
但是语句不能用作表达式，不如赋值语句不能嵌入到其他表达式。
这样做的理由是避免常见的编写错误。
比如 `while` 循环里做相等性测试时可以防止把 `==` 写成 `=`。

只调用原位置修改操作，而不要赋值调用的结果。

```
>>> L = L.append(4) # But append returns None, not L
>>> print(L) # So we lose our list!
None
```

11.3 打印操作

`print` 将一个或多个对象转换成相应的文本表示，添加一些格式，发送最终结果给标准输出或其他类文件的流 `file-like stream`。

打印 `print` 和 `file.write` 方法的不同之处在于，默认写入对象到 `stdout` 流，同时加入一些格式化，不需要预先把对象转换为字符串。

标准输出流 `stdout` 是程序文本输出的默认发送地。
当脚本启动时，会自动创建 `3` 个数据连接，就是标准输出流，标准输入流和标准出错流。

> 标准错误流好一点吧。

标准输出流通常映射到 `Python` 程序的启动窗口，除非它已在操作系统的 `shell` 中被重定向到一个文件或管道。

由于标准输出流在 `Python` 中可以作为内置的 `sys` 模块中的 `stdout` 文件对象来使用，因此你也可以用文件的写入方法调用来模拟 `print`。然而，你也可以借助 `print`，把文本打印到其他文件或流。

```
>>> import sys
```

```
>>> res = sys.stdout.write('你好啊\n')
你好啊
>>> res
4

# 等价于
>>> print('你好啊')
你好啊
```

可以在 `2.X` 中导入和使用 `3.X` 风格的打印。

`3.X` 中的 `print` 函数

`print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout][, flush=False])`
其中 `flush` 为 `3.3` 新增
`print` 内置函数打印了一个或多个对象的文本表示，在中间用字符串 `sep` 来分隔，在结尾加上字符串 `end`，通过 `file` 来指定输出流，并按照 `flush` 来决定是否刷新输出缓冲区。

必须使用关键字参数指定 `sep`，`end` 和 `file`。
`file` 参数可以传入任何带有一个类似文件的 `write(string)` 方法的对象。真正的文件应该已经为输出打开。

打印内容是否被缓冲在内存中，是由 `file` 决定的，传入 `True` 会强制刷新输出流。

`print` 在内部实现上等价于把待打印对象传入 `str`。（技术上说，并不是真的这样做，但是效果是一样的）

```
>>> print(x, y, z, sep='...', file=open('data.txt', 'w')) # Print to a fi
le
>>> print(x, y, z) # Back to stdout
spam 99 ['eggs']
>>> print(open('data.txt').read()) # Display file text
spam...99...['eggs']
```

`2.X` 的 `print` 语句

Table 11-5. Python 2.X print statement forms

Python 2.X statement	Python 3.X equivalent	Interpretation
`print x, y`	`print(x, y)`	Print objects' textual forms to `sys.stdout`; add a space between the items and an end-of-line at the end
`print x, y,`	`print(x, y, end='')`	Same, but don't add end-of-line at end of text
`print >> afile, x, y`	`print(x, y, file=afile)`	Send text to `afile.write`, not to `sys.stdout.write`

```
>>> import sys # Printing the hard way
>>> sys.stdout.write('hello world\n')
hello world
```

也就是说：
`print(X, Y) # Or, in 2.X: print X, Y`
等价于

```
import sys
sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
```

手动重定向输出流

```
import sys
sys.stdout = open('log.txt', 'a') # Redirects prints to a file
...
print(x, y, x) # Shows up in log.txt
```

通过赋值 `sys.stdout` 会让程序中所有的 `print` 都被重新定向。
`print` 语句会很乐意持续地调用 `sys.stdout` 的 `write` 方法，而不管 `sys.stdout` 引用的是什么。
你甚至可以将 `sys.stdout` 重设为非文件的对象，只要该对象满足预期的协议，即拥有 `write` 方法。

切换输出流的定向

```
C:\code> c:\python33\python
>>> import sys
>>> temp = sys.stdout                    # Save for restoring later
>>> sys.stdout = open('log.txt', 'a')    # Redirect prints to a file
>>> print('spam')                        # Prints go to file, not here
>>> print(1, 2, 3)
>>> sys.stdout.close()                   # Flush output to disk
>>> sys.stdout = temp                    # Restore original stream

>>> print('back here')                   # Prints show up here again
back here
>>> print(open('log.txt').read())        # Result of earlier prints
spam
1 2 3
```

还可以使用 `sys.__stdout__` 属性来让 `sys.stdout` 恢复。

单次 `print` 到文件：

```
log = open('log.txt', 'a')      # 3.X
print(x, y, z, file=log)        # Print to a file-like object
print(a, b, c)                  # Print to original stdout

log = open('log.txt', 'a')      # 2.X
print >> log, x, y, z           # Print to a file-like object
```

```
print a, b, c                   # Print to original stdout
```

可以使用 `print` 把信息打印到标准错误流。

```
>>> import sys
>>> sys.stderr.write(('Bad!' * 8) + '\n')
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!

# 等价于
>>> print('Bad!' * 8, file=sys.stderr) # In 2.X: print >> sys.stderr, 'Bad!' * 8
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
```

打印 `print` 和文件的 `write` 方法的等价性不言而喻。

版本中立的打印

- 使用 `2to3` 转换器，或者 `3to2` 转换器。
- 从 `__future__` 导入
  在 `2.X` 中 `from __future__ import print_function`。这条语句让 `2.X` 支持 `3.X` 的 `print` 函数，可以直接在 `2.X` 中写 `3.X` 的形式。
  注意：
  - 在 `3.X` 中这条语句直接被忽略。
  - 这条语句仅为 *单个文件* 修改解析器，所以它要出现在每一个包含打印语句的 `2.X` 文件中，仅仅从外部导入是不够的。
- 用代码消除显示差别
  可以在 `2.X` 的打印语句外面加一个圆括号。
  缺点是如果有多个或者零个待打印对象时，`2.X` 会产生一个元组，额外打印一对圆括号。

```
C:\code> c:\python33\python
>>> print('spam') # 3.X print function call syntax
spam
>>> print('spam', 'ham', 'eggs') # These are multiple arguments
spam ham eggs

# 对比 Python 2
c:\code> py -2
>>> print() # This is just a line-feed on 3.X

>>> print('') # This is a line-feed in both 2.X and 3.X
```

> 【注】所以如果要通用地打印空行，使用 `print('')` 比 `print()` 要好。

`2.X` 中元组内的字符串也被引号包围。这是因为当一个对象嵌套在另一个对象中时，与单独作为顶层对象相比，打印方式可能有所不同。
技术上讲，嵌套效果会使用 `repr` 显示，而顶层对象会使用 `str` 显示。

如果不想全局开启 `3.X` 的打印，还想避免嵌套带来的显示差异，可以将待打印字符串 处理成一个单独的对象 来实现版本统一。

```
>>> print('%s %s %s' % ('spam', 'ham', 'eggs'))
spam ham eggs
>>> print('{0} {1} {2}'.format('spam', 'ham', 'eggs'))
spam ham eggs
>>> print('answer: ' + str(42))
answer: 42
```

请留意：`print` 和 `stdout`
理解 `print` 语句和 `sys.out` 之间的等价性是相当重要的。
这也是为什么我们可以把 `sys.out` 重新赋值给用户自定义（能够提供 `write` 方法的）的类似文件的对象。

```
class FileFaker:
    def write(self, string):
        # Do something with printed text in string

import sys
sys.stdout = FileFaker()
print(someObjects) # Sends to class write method
```

这能行得通是因为 `print` 是一个多态运算 `polymorphic operation`，它不管 `sys.stdout` 到底是什么，只要它有一个 `write` 方法（接口）即可。

> 【注】鸭子类型？

`it doesn't care what sys.stdout is, only that it has a method (i.e., interface) called write.`

使用 `3.X` 的 `file` 参数，而 `2.X` 的 `>>`，都让重定向变的更加简单，不需要显式地重新设置 `sys.stdout`。

```
myobj = FileFaker()              # 3.X: Redirect to object for one print
print(someObjects, file=myobj)   # Does not reset sys.stdout

myobj = FileFaker()              # 2.X: same effect
print >> myobj, someObjects      # Does not reset sys.stdout
```

`3.X` 的 `input` 函数和 `2.X` 的 `raw_input` 函数会从 `sys.stdin` 文件读入，所以可以用类似的方式拦截对读取的请求：即使用类实现类似文件的 `read` 方法。

`Python` 的打印操作重定向工具，实质上是 `shell` 脚本语法在 `Python` 中的替身。

11.4 本章小结

11.5 本章习题

1. 举出三种可以让三个变量赋值成相同值的方式。
   多目标赋值语句 `a = b = c = 0`
   序列赋值语句 `a, b, c = 0, 0, 0`
   单独赋值语句 `a = 0; b = 0; c = 0`
2. 怎样使用 `print` 语句来向外部文件发送文本？
   - 单次打印操作可以用 `3.X` 的 `print(x, file=f)`
   - 或者用 `2.X` 的 `print >> f, x`
   - 或者直接给 `sys.stdout` 赋值为手动打开的文件，然后恢复初始值
   - 用系统的 `shell` 的特殊语法，把程序所有的打印文字重定向到一个文件，但这是 `Python` 范围以外的内容了

---

12. if 测试和语法规则

12.1 if 语句

多路分支
`Python` 中没有 `switch` 或 `case` 语句。
多路分支在 `Python` 中写成一系列的 `if/elif` 测试，或者偶尔采取索引字典和查询列表的形式。因为字典和里诶包可以在运行时动态地创建，所以有时候会比在脚本中硬编码的 `if` 逻辑更有灵活性。

```
>>> choice = 'ham'
>>> print({'spam': 1.25, # A dictionary-based 'switch'
...        'ham': 1.99, # Use has_key or get for default
...        'eggs': 0.99,
...        'bacon': 1.10}[choice])
1.99
```

使用 `if` 的多路分支看起来比较冗长，灵活性差，但是可读性好：

```
>>> if choice == 'spam': # The equivalent if statement
...     print(1.25)
... elif choice == 'ham':
...     print(1.99)
```

```
... elif choice == 'eggs':
...     print(0.99)
... elif choice == 'bacon':
...     print(1.10)
... else:
...     print('Bad choice')
...
1.99
```

处理选择语句的默认情况
使用字典的 `get` 方法

```
>>> branch = {'spam': 1.25,
...           'ham': 1.99,
...           'eggs': 0.99}
>>> print(branch.get('spam', 'Bad choice'))
1.25
>>> print(branch.get('bacon', 'Bad choice'))
Bad choice
```

等效的方法是用 `if` 配合 `in` 成员测试：

```
>>> choice = 'bacon'
>>> if choice in branch:
...     print(branch[choice])
... else:
...     print('Bad choice')
...
Bad choice
```

还可以使用 `try` 捕获异常：

```
>>> try:
...     print(branch[choice])
... except KeyError:
...     print('Bad choice')
...
Bad choice
```

处理更复杂的行为
字典可以包含函数，从而实现一般化的跳转表 `jump table`。
这些函数作为字典的值，通常写成函数名或者一行 `lambda` 表达式，通过 *添加括号调用* 来触发动作。

```
def function(): ...
def default(): ...
```

```
branch = {'spam': lambda: ...,     # A table of callable function objects
          'ham': function,
          'eggs': lambda: ...}

branch.get(choice, default)()
```

基于字典的多路分支在处理动态数据时很有用，更多的程序员认为编写 `if` 语句更加直接。

12.2 复习 Python 语法规则

语句是逐个运行的，除非你编写了其他内容。
`Python` 执行程序的路径被称为控制流 `control flow`，所以像 `if` 这样会改变控制流的语句常被称为 `control-flow statement` 控制流语句。

块和语句的边界会自动被解释器识别。

复合语句 = 首行 + “:” + 多个缩进语句。
缩进语句称为块 `block`，有时候称为组 `suite`。
如果语句块是简单的非复合语句，那么它可以与首行放在同一行。

空白行、空格以及注释通常都会被忽略。
空白行在交互式命令行下不会被忽略，而是被用作结束复合语句。
除了缩进中以及在字符串字面量中的空格之外，语句和表达式中的空格几乎都会被忽略。

文档字符串(docstring)会被忽略，但是会被保存并由工具显示。

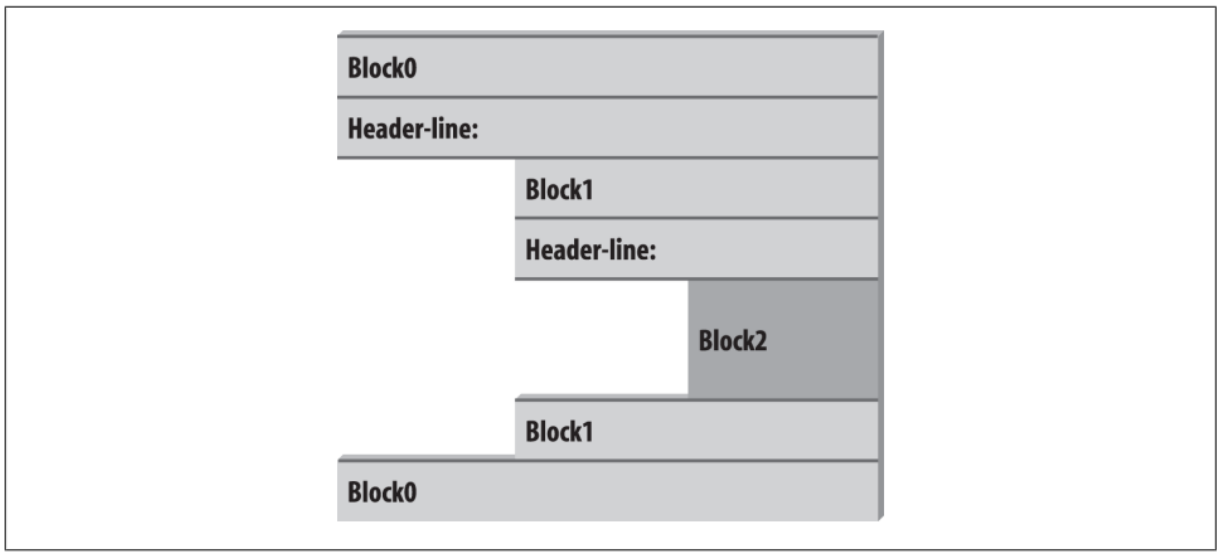


Figure 12-1. Nested blocks of code: a nested block starts with a statement indented further to the right and ends with either a statement that is indented less, or the end of the file.

代码块分隔符：缩进规则
通常来说，顶层（无嵌套）代码必须开始于第 `1` 列。

嵌套块可以从任何列开始。
缩进可以由任意的空格和制表符组成，只要单个块中的所有语句都相同即可。
缩进确实是 `Python` 语法中的一部分，而不仅是编程风格。

避免混合使用制表符和空格，`3.X` 会报错
`2.X` 不会报错，但是有一个 `-t` 命令行参数，辅助检测制表符用法上的不一致。
还有一个 `-tt` 参数会对这样的代码进行报错，它等同于 `3.X` 的报错情况。
使用方法：`python -t main.py`

语句分隔符：行间连接符

- 如果使用语法括号对，语句就可以横跨数行。
  语句的续行可以从任何缩进层次开始，但你应该尽可能让它们垂直对齐以便于阅读。
- 如果语句以反斜杠结尾，就可横跨数行。
  这是一个有点过时的功能，一般并不推荐大家使用。
  `\` 不能被嵌入到字符串字面量或注释
- 字符串字面量有特殊规则。
  三重引号字符串块可以横跨多行。
  相邻的字符串字面量会被隐式拼接。与圆括号共同使用，就可以让多个字符串横跨多行。

```
>>> ('我'
...  '爱'
...  '你'
...  '中'
...  '国')
'我爱你中国'
```

- 其他规则
  不常见：用分号终止语句，用来把一个以上的简单（非复合）语句挤进同一行中。

现在不喜欢用反斜杠的一个原因是，万一漏了，有时候不会报错。

```
x = 1 + 2 + 3 \ # Omitting the \ makes this very different!
+4
```

把 `\naaaa\nbbbb\ncccc` 赋值给 `S`

```
S = """
aaaa
bbbb
cccc"""
```

把 `aaaabbbbcccc` 赋值给 `S`

```
S = ('aaaa'
     'bbbb'      # Comments here are ignored
     'cccc')
```

12.3 真值和布尔测试

- 所有对象都有一个固有的布尔值
- 任何非零数字或非空对象都为真
- 数字零、空对象以及特殊对象 `None` 都被认作是假
- 比较和相等测试会递归地应用到数据结构中
- 比较和相等测试会返回 `True` 或 `False`（ `1` 和 `0` 的特殊版本）
- 布尔 `and` 和 `or` 运算符会返回真或假的操作数对象

  > ? ? ?

- 布尔运算符会在结果确定的时候立即停止计算（短路）

`Python` 有三种布尔表达式运算符：（不是 `C` 语言的 `&&`，`||`，`!`）

- `X and Y`
- `X or Y`
- `not X`

布尔 `and` 和 `or` 运算符在 `Python` 中会返回对象，而不是值 `True` 或 `False`。要么是运算符左侧的对象，要么是右侧的对象。

```
>>> 2 or 3, 3 or 2       # Return left operand if true
(2, 3)                   # Else, return right operand (true or false)
>>> [] or 3
3
>>> [] or {}
{}

>>> 2 and 3, 3 and 2     # Return left operand if false
(3, 2)                   # Else, return right operand (true or false)
>>> [] and {}
[]
>>> 3 and []
[]
```

12.4 if/else 三元表达式

`A = Y if X else Z`
等价于

```
if X:
    A = Y
else:
    A = Z
```

`2.5` 之前的版本：
`A = ((X and Y) or Z)`
注意：不完全等价，要假设 `Y` 为布尔真值。

还有一种 完全 等价版本：
`A = [Z, Y][bool(X)]`
比如

```
>>> ['f', 't'][bool('')]
'f'
>>> ['f', 't'][bool('spam')]
't'
```

请留意：布尔值
`Python` 中的布尔运算符有一些不寻常的作用。
`X = A or B or C or None`
通过 `or` 从一组对象中做选择。把 `X` 设为 `A`、`B`、`C` 中的第一个非空（为真）的对象；如果所有对象都为空，就设为 `None`。

这样行得通是因为 `or` 于是那UN福会返回其左右的两个对象之一，这成为 `Python` 中相当常见的代码编写技巧。
从一个固定大小的集合中选择非空的对象，只要把它们串在一个 `or` 表达式中即可。
`X = A or default`

`or` 还可以与 `if/else` 配合，避免一些不必要的代码运行。

```
if f1() or f2(): ...
```

如果 `f1()` 返回值真值（非空），那么 `Python` 永远不会执行 `f2`。
如果想要两个函数都执行，需要在 `or` 之前调用它们。

```
tmp1, tmp2 = f1(), f2()
if tmp1 or tmp2: ...
```

在 `Python` 中，直接测试对象 `if X:` 比空值比较 `if X != '':` 更为常见和简单。

使用类定义新的类型时，可以用 `__bool__` 或 `__len__` 方法指定它的布尔特性。
`Python 2.7` 中 `__bool__` 叫做 `__nonzero__`。

如果 `__bool__` 没有被重载的话，`__len__` 就会被调用并返回（等效于空对象为假）

与 `or` 链式求值 类似的工具。
`filter` 函数、列表推导、`any` 和 `all`

```
>>> L = [1, 0, 2, 0, 'spam', '', 'ham', []]
>>> list(filter(bool, L)) # Get true values
[1, 2, 'spam', 'ham']
>>> [x for x in L if x] # Comprehensions
[1, 2, 'spam', 'ham']
>>> any(L), all(L) # Aggregate truth
(True, False)
```

12.5 本章小结

`if` 语句，是第一个复合及逻辑语句。

12.6 本章习题

---

13. while 循环和 for 循环

13.1 while 循环

`else` 块会在控制权离开循环而又没有碰到 `break` 语句时执行。（碰到 `continue` 不影响）

跳出死循环用 `ctrl + c`

`Python` 中没有某些语言中所谓的 `do until` 循环语句。
不过可以用下面的方式实现，保证循环体至少执行一次：

```
while True:
    ...loop body...
    if exitTest(): break
```

13.2 `break`、`continue`、`pass` 和循环的 `else`

`pass` 是无运算的占位语句，当语法要求有一条语句却没有任何实际的语句可写时，就可以使用它。
它通常为复合语句编写一个空的主体。

应用场景：

- 忽略 `try` 语句所捕获的异常
- 定义空的类对象，用于携带属性并扮演其他编程语言中 “结构体” 或 “记录” 的角色。

`pass` 时常表示 “以后会填上”，也就是暂时填充函数的主体。

`3.X` 允许在可以使用表达式的任何地方使用 `...` 来省略代码。
可以当做是 `pass` 语句的一种替代方案，尤其是对于随后填充的代码。

> 【注】：`ellipsis` 省略号，复数是 `ellipses`

比如：

```
def func1():
    ...                 # Alternative to pass
def func2():
    ...
func1()                 # Does nothing if called
```

`...` 也可以和语句头部出现在同一行，还可以初始化变量名（代替 `None`）。

```
def func1(): ...        # Works on same line too
def func2(): ...

>>> X = ...             # Alternative to None
>>> X
Ellipsis
```

`...` 的最初意图是切片扩展 `slice extension`。
能否推广开来与 `pass` 和 `None` 的这类用法相抗衡，还需拭目以待。

> 最初意图到底是啥？这里也没说。

`input` 也会在用户键入文件结束符，例如在 `Windows` 上按下 `Ctrl+Z` 组合键或在 `Unix` 上按下 `Ctrl+D` 组合键时引发异常。如果要考虑文件结束符的情况，可以用 `try` 语句把 `input` 括起来。

如果循环主体从来没有执行过，循环 `else` 分句也会执行，因为此时你也没有在循环体执行 `break` 语句。
举例：判断质数

```
x = y // 2                          # For some y > 1
while x > 1:
    if y % x == 0:                  # Remainder
        print(y, 'has factor', x)
        break                       # Skip else
    x -= 1
else:                               # Normal exit
    print(y, 'is prime')
```

简而言之，循环 `else` 分句就是提供了常见代码编写情形的显式语法：这是让你不必设置和检查标志位，就能够捕捉循环退出情况的一种编程结构。
当与 `for` 循环组合使用时，循环 `else` 分句会变得更加有用，因为序列迭代是不受你控制的。

请留意：仿真 `C` 语言的 `while` 循环

```
while ((x = next(obj)) != NULL) {...process x...}
```

`C` 语言的赋值运算会返回被赋予的值，但 `Python` 的赋值语句却只是语句，而不是表达式。
这样就消除了一个众所周知的 `C` 语言错误：当需要使用 `==` 的时，在 `Python` 中是不会被不小心打成 `=` 的。
如果需要，有三种方式可以实现 `C` 语言中的效果：

- 配合 `break`，把赋值语句移入循环体：

```
while True:
    x = next(obj)
    if not x: break
    ...process x...
```

- 配合 `if` 测试，把赋值语句移入循环体：

```
x = True
while x:
    x = next(obj)
    if x:
        ...process x...
```

- 还可以把第一次赋值移到循环体外

```
x = next(obj)
while x:
    ...process x...
```

```
        x = next(obj)
```

有些人认为第一种是最不结构化的，但这也似乎是最简单、最常用的。

简单的 `for` 循环可以取代这样的 `C` 语言循环：

```
for x in obj: ...process x...
```

13.3 for 循环

`for` 循环在 `Python` 中是一个通用的迭代器：它可以遍历任何有序序列或其他可迭代对象内的元素。

> 这里的迭代器的意思跟 `Python` 中的迭代器 `iterator` 还不太一样。

`Python` 的 `for` 循环首行指定了一个（或一些）赋值目标，以及你想遍历的对象。首行后面是你想重复的语句块（需要缩进）。

```
>>> for x in ["spam", "eggs", "ham"]:
...     print(x, end=' ')
...
spam eggs ham
```

`for` 循环可用于字符串和元组。

`for` 循环中的元组赋值

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T: # Tuple assignment at work
...     print(a, b)
...
1 2
3 4
5 6
```

这种方式通常与 `zip` 一起使用，实现并行遍历 `parallel traversal`。
还经常与 `SQL` 数据库一起使用，其中查询结果表作为像这里的列表一样的序列返回——外层的列表就是数据库表，内嵌的元组就是表中的行，并通过元组赋值提取每一行的信息。

`for` 配合字典的 `items()` 方法遍历键和值，而不用再遍历键并手动索引获取值。

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
```

```
>>> for key in D:
...     print(key, '=>', D[key])     # Use dict keys iterator and index
...
a => 1
c => 3
b => 2

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> for (key, value) in D.items():
...     print(key, '=>', value)      # Iterate over both keys and values
...
a => 1
c => 3
b => 2
```

还可以手动在循环内解包：

```
>>> T
[(1, 2), (3, 4), (5, 6)]

>>> for both in T:
...     a, b = both     # Manual assignment equivalent
...     print(a, b)     # 2.X: prints with enclosing tuple "()"
...
1 2
3 4
5 6
```

嵌套的结构也可以解包：

```
>>> ((a, b), c) = ((1, 2), 3)    # Nested sequences work too
>>> a, b, c
(1, 2, 3)

>>> for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: print(a, b, c)
...
1 2 3
4 5 6
```

任何嵌套的序列都可以以这种方式解包：

```
>>> for ((a, b), c) in [([1, 2], 3), ['XY', 6]]: print(a, b, c)
...
1 2 3
X Y 6
```

`for` 循环中的 `Python 3.X` 扩展序列赋值

```
>>> for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
...     print(a, b, c)
...
1 [2, 3] 4
5 [6, 7] 8
```

在 `2.X` 中可以通过切片来实现类似的效果。

```
>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:     # Manual slicing in 2.X
...     a, b, c = all[0], all[1:3], all[3]
...     print(a, b, c)
...
1 (2, 3) 4
5 (6, 7) 8
```

切片和星号解包的区别在于，切片返回的是一个相应数据类型的结果，而星号名称总会被返回成列表。

一般来说，为了获得更简洁的代码和更优异的性能，你应该把尽可能多的工作交给 `Python` 来完成。

> 所以要尽量用内置的函数，不要自己造轮子

嵌套 `for` 循环
例子：找出相同的字符放到一个列表中。

```
>>> seq1 = "spam"
>>> seq2 = "scam"
>>>
>>> res = [] # Start empty
>>> for x in seq1: # Scan first sequence
...     if x in seq2: # Common item?
...         res.append(x) # Add to result end
...
>>> res
['s', 'a', 'm']
```

使用列表推导式更简单：

```
>>> [x for x in seq1 if x in seq2] # Let Python collect results
['s', 'a', 'm']
```

请留意：文件扫描器
一次性加载文件内容到字符串并打印

```
file = open('test.txt', 'r')     # Read contents into a string
print(file.read())
```

分块加载文件，按字符读取

```
file = open('test.txt')

# while 循环版
while True:
    char = file.read(1) # Read by character
    if not char: break # Empty string means end-of-file
    print(char)

# for 循环版
for char in open('test.txt').read():
    print(char)
```

上面的 `for` 循环版仍然是一次性加载到内存。
如果要按行或按块读取：

```
# while 循环版
file = open('test.txt')
while True:
    line = file.readline() # Read line by line
    if not line: break
    print(line.rstrip()) # Line already has a \n
    # 或者写成 print(line, end='') 也可以

# for 循环版
file = open('test.txt', 'rb')
while True:
    chunk = file.read(10) # Read byte chunks: up to 10 bytes
    if not chunk: break
    print(chunk)
```

你通常会按块读取二进制数据。
不过如果要逐行读取文本文件，那么 `for` 循环最好用：

```
for line in open('test.txt').readlines():
    print(line.rstrip())

for line in open('test.txt'): # Use iterators: best for text input
    print(line.rstrip())
```

最后一种方式通常是文本文件的最佳选择！
对内存压力小，迭代器速度也很快。

`readlines` 方法也很有用，比如需要按行反转一个文件。
注意：`reversed` 接收一个序列，而不是可迭代对象。

> 这跟 `sorted` 不一样，`sorted` 接收可迭代对象。

```
for line in reversed(open('test.txt').readlines()): ...
```

`2.X` 中可能看到用 `file` 来打开文件和使用 `xreadlines` 方法。

13.4 编写循环的技巧

`for` 循环通常比 `while` 循环更容易写，也执行得更快。
所以 `for` 循环一般是你遍历序列或其他可迭代对象时的首选。

一套内置函数，可以帮你在 `for` 循环内定制迭代：

- 内置函数 `range`（`Python 0.X` 及之后版本可用），返回一系列连续增加的整数，可作为 `for` 中的索引。
- 内置函数 `zip`（`Python 2.0` 及之后版本可用）返回一系列并行元素的元组，可用于在 `for` 内遍历多个序列。
- 内置函数 `enumerate`（`Python 2.3` 及之后版本可用）同时生成可迭代对象中元素的值和索引，因而我们不必再手动计数。
- 内置函数 `map`（`Python 1.0` 及之后版本可用）在 `Python 2.X` 中与 `zip` 有相似的效果，但是在 `3.X` 中 `map` 的这一角色被移除了。

`2.X` 中的 `xrange` 类似 `3.X` 中的 `range`。

> **注意**：`range` 得到的是一个可迭代对象，但不是迭代器。

```
>>> r = range(3)
>>> r is iter(r)
False
>>> it = iter(r)
>>> it is iter(it)
True
>>>
```

一条通用准则：尽可能使用 `for` 而不是 `while`，并且不要在 `for` 循环中使用 `range` 调用。

序列乱序器 `sequence shuffler`

```
>>> S = 'spam'
>>> for i in range(len(S)): # For repeat counts 0..3
...     S = S[1:] + S[:1] # Move front item to end
...     print(S, end=' ')
...
pams amsp mspa spam

>>> S
'spam'
>>> for i in range(len(S)): # For positions 0..3
...     X = S[i:] + S[:i] # Rear part + front part
...     print(X, end=' ')
...
spam pams amsp mspa
```

这种操作还适用于其他序列：

```
>>> L = [1, 2, 3]
>>> for i in range(len(L)):
...     X = L[i:] + L[:i]     # Works on any sequence type
...     print(X, end=' ')
...
[1, 2, 3] [2, 3, 1] [3, 1, 2]
```

非穷尽遍历

```
>>> S = 'abcdefghijk'
>>> list(range(0, len(S), 2))
[0, 2, 4, 6, 8, 10]

>>> for i in range(0, len(S), 2): print(S[i], end=' ')
...
a c e g i k
```

但是最佳实践是 使用切片表达式的三参数形式。

```
>>> S = 'abcdefghijk'
>>> for c in S[::2]: print(c, end=' ')
...
a c e g i k
```

这里 `range` 的优点是不占用空间，而切片会复制字符串。

修改列表

需求：让列表中的每个元素都加 `1`。

```
>>> L = [1, 2, 3, 4, 5]

>>> for x in L:
...     x += 1 # Changes x, not L
...
>>> L
[1, 2, 3, 4, 5]
>>> x
6
```

这里修改的是循环变量 `x` 而不是列表 `L`。
要遍历同时修改列表，需要借助索引。

```
>>> L = [1, 2, 3, 4, 5]

>>> for i in range(len(L)):         # Add one to each item in L
...     L[i] += 1                   # Or L[i] = L[i] + 1
...
>>> L
[2, 3, 4, 5, 6]
```

等价 `while` 形式：

```
>>> i = 0
>>> while i < len(L):
...     L[i] += 1
...     i += 1
...
>>> L
[3, 4, 5, 6, 7]
```

最好的写法：

```
[x + 1 for x in L]  # 这样新生成了一个列表，没有对原列表进行原位置修改
```

并行遍历 `zip` 和 `map`
与 `range` 一样，`zip` 在 `Python 2.X` 中返回一个列表，但在 `Python 3.X` 中则返回一个可迭代对象，我们必须将其包含在一个 `list` 调用中才能一次性显示所有结果。

```
>>> L1 = [1,2,3,4]
>>> L2 = [5,6,7,8]

>>> for (x, y) in zip(L1, L2):
...     print(x, y, '--', x+y)
...
1 5 -- 6
```

```
2 6 -- 8
3 7 -- 10
4 8 -- 12
```

最终结果就是在循环中同时扫描了 `L1` 和 `L2`。
`zip` 接收任何类型的可迭代对象，并且支持两个以上的参数。

> 仅传入一个对象也可以，会形成一个个的单元素元组。

```
>>> L = range(3)
>>> list(zip(L))
[(0,), (1,), (2,)]
```

如果向 `zip` 输入 `N` 个参数，我们将得到 `N` 元素元组的一个可迭代对象。

`zip` 会以最短序列的长度为准来截断结果元组。

`2.X` 中 `map` 的等价形式
`map` 在传入 `None` 作为函数参数时，会用类似的方式把序列的元素配对。
如果各个参数长度不等，会用 `None` 补齐较短的序列。

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'

>>> map(None, S1, S2)     # 2.X only: pads to len(longest)
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'),
(None,'3')]
```

`3.X` 中的 `map`

```
>>> list(map(ord, 'spam'))
[115, 112, 97, 109]
```

`map` 比下面的等价形式要快

```
>>> res = []
>>> for c in 'spam': res.append(ord(c))
>>> res
[115, 112, 97, 109]
```

使用 `zip` 构造字典
当键和值的集合必须在运行时计算时，`zip` 可以很方便地生成字典。

```
>>> keys = ['spam', 'eggs', 'toast']
```

```
>>> vals = [1, 3, 5]

>>> list(zip(keys, vals))
[('spam', 1), ('eggs', 3), ('toast', 5)]

>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v
...
>>> D2
{'eggs': 3, 'toast': 5, 'spam': 1}
```

`2.2` 和后续版本，可以省掉 `for` ：

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]

# 本质上是一个对象构造的请求
>>> D3 = dict(zip(keys, vals))
>>> D3
{'eggs': 3, 'toast': 5, 'spam': 1}
```

`dict` 实际上是一个类型名称

> 类型和类在 `3.X` 统一了，所以 `dict` 也是类名，所以类名不一定都是首字母大写。

字典推导：

```
>>> {k: v for (k, v) in zip(keys, vals)}
{'eggs': 3, 'toast': 5, 'spam': 1}
```

同时给出偏移量和元素：`enumerate`
可以维护一个当前偏移量的计数器。

```
>>> S = 'spam'
>>> offset = 0
>>> for item in S:
...     print(item, 'appears at offset', offset)
...     offset += 1
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

或者使用 `enumerate`

```
>>> S = 'spam'
>>> for (offset, item) in enumerate(S):
...     print(item, 'appears at offset', offset)
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

`enumerate` 函数返回一个 *生成器对象*，这种对象支持迭代协议。
生成器对象有一个方法可以被 `next()` 内置函数调用。

```
>>> E = enumerate(S)
>>> E
<enumerate object at 0x0000000002A8B900>
>>> next(E)
(0, 's')
>>> next(E)
(1, 'p')
>>> next(E)
(2, 'a')
```

所有的 迭代上下文（包括列表推导）都会自动执行迭代协议。

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'p', 'aa', 'mmm']

>>> for (i, l) in enumerate(open('test.txt')):
... print('%s) %s' % (i, l.rstrip()))
...
0) aaaaaa
1) bbbbbb
2) cccccc
```

> **注意**
> 这里有意思的是，`0 * 字符串` 得到的是一个空字符串！
> 经试验，`<=0` 的数乘以字符串，得到的都是空字符串！

请留意：`shell` 命令及其他
`os.popen` 方法也给出了一个类文件接口，可以读取被触发的 `shell` 命令的输出。
需求：运行 `shell` 命令并读取其标准输出文本。
你需要将该 `shell` 命令作为一个字符串传入 `os.popen`，然后从它返回的类文件对象中读取文本。

```
>>> import os
>>> F = os.popen('dir') # Read line by line
```

```
>>> F.readline()
' Volume in drive C has no label.\n'
>>> F = os.popen('dir') # Read by sized blocks
>>> F.read(50)
' Volume in drive C has no label.\n Volume Serial Nu'

>>> os.popen('dir').readlines()[0] # Read all lines: index
' Volume in drive C has no label.\n'
>>> os.popen('dir').read()[:50] # Read all at once: slice
' Volume in drive C has no label.\n Volume Serial Nu'

>>> for line in os.popen('dir'): # File line iterator loop
...     print(line.rstrip())
...
Volume in drive C has no label.
Volume Serial Number is D093-D1F7
...and so on...
```

任何一条通过命令行启动的程序，都可以用这种方式来开启。
还有一种方式：`os.system` 直接运行一条 `shell` 命令，`os.popen` 则将 `Python` 程序连通到这条 `shell` 命令的输出流。

> 网上的说法是，`os.system` 返回脚本的 *退出状态码*，如果为 `0`，表示命令执行成功，否则出错。`os.popen` 返回的是脚本执行过程中的输出内容，是一个类文件对象。

对比一下两种方式

```
>>> os.system('systeminfo')
...output in console, popup in IDLE...
0

>>> for line in os.popen('systeminfo'): print(line.rstrip())
Host Name: MARK-VAIO
OS Name: Microsoft Windows 7 Professional
OS Version: 6.1.7601 Service Pack 1 Build 7601
...lots of system information text...
```

像 `os.popen` 和 `os.system` 以及 `subprocess` 模块这些工具允许你使用自己计算机上的每一条命令行程序，但你也可以使用 `Python` 代码来编写模拟那些命令行程序。

比如模拟 `Unix` 的 `awk` 工具，提取文本第 `7` 列：

```
# awk emulation: extract column 7 from whitespace-delimited file
for val in [line.split()[6] for line in open('input.txt')]:
    print(val)

# Same, but more explicit code that retains result
col7 = []
```

```
for line in open('input.txt'):
    cols = line.split()
    col7.append(cols[6])
for item in col7: print(item)

# Same, but a reusable function (see next part of book)
def awker(file, col):
    return [line.rstrip().split()[col-1] for line in open(file)]

print(awker('input.txt', 7))            # List of strings
print(','.join(awker('input.txt', 7)))  # Put commas between
```

`Python` 的类文件接口：

- 网站返回的文本
- 由 `URL` 标识的网页

等等

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://home.rmi.net/~lutz'):
...     print(line)
...
b'<HTML>\n'
b'\n'
b'<HEAD>\n'
b"<TITLE>Mark Lutz's Book Support Site</TITLE>\n"
...etc...
```

`2.X` 中要使用 `urllib` 而不是 `urllib.request`。而且 `2.X` 的返回值是文本字符串。

13.5 本章小结

13.6 本章习题

1. `while` 和 `for` 之间主要的功能性区别是什么？
   `while` 循环是一种通用的循环语句，`for` 循环被设计用来在一个序列或其他可迭代对象中遍历各项。尽管 `while` 可以用计数器来模拟 `for` 循环，但它需要更多的代码并且运行起来可能更慢。
2. `range` 在 `for` 循环中有哪些用法？
   - 实现指定次数的重复
   - 按照偏移量而不是偏移量处的元素来扫描
   - 在过程中按一定步长跳过元素
   - 遍历一个列表的时候修改它

注意：还有更好的替代方案，比如三参数切片、列表推导。

---

14. 迭代和推导

14.1 迭代器：初次探索

`for` 循环可以用于任何可迭代对象。
实际上，对 `Python` 中所有能够从左至右扫描对象的迭代工具而言都是如此。
比如：

- `for` 循环
- 列表推导
- `in` 成员测试（`membership test`）
- 内置函数 `map` 等

关于可迭代对象 `iterable` 和迭代器 `iterator` 的术语不是很严格。
本书这样规定：

- 可迭代对象是可以被 `iter` 调用的对象
- 迭代器是由 `iter(可迭代对象)` 返回的可以被 `next` 调用的对象

文件对象有一个 `readline` 方法，到达文件末尾时，就会返回空字符串。
而 `3.X` 的文件还有一个 `__next__` 方法，每次调用，就会返回文件的下一行，唯一和 `readline` 的区别就是，到达文件末尾时，会触发内置的 `StopIteration` 异常，而不是返回空字符串。

```
>>> f = open('script2.py')   # __next__ loads one line on each call too
>>> f.__next__()             # But raises an exception at end-of-file
'import sys\n'
>>> f.__next__()             # Use f.next() in 2.X, or next(f) in 2.X or
 3.X
'print(sys.path)\n'
>>> f.__next__()
'x = 2\n'
>>> f.__next__()
'print(x ** 32)\n'
>>> f.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

最简单的写法，运行最快，内存占用最小：

```
>>> for line in open('script2.py'):  # Use file iterators to read by line
s
```

```
...     print(line.upper(), end='')  # Calls __next__, catches StopIterat
ion
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(X ** 32)
```

`while` 循环版本：

```
>>> f = open('script2.py')
>>> while True:
...     line = f.readline()
...     if not line: break
...     print(line.upper(), end='')
...
...same output...
```

`while` 循环会比基于迭代器的 `for` 循环运行得更慢，因为迭代器在 `Python` 内部是以 `C` 语言速度运行的，而 `while` 循环版本则是通过 `Python` 的虚拟机运行 `Python` 字节码的。
任何时候，我们把 `Python` 代码换成 `C` 程序代码，速度都会有所提升。（并不绝对，尤其是在 `3.X` 中）

`2.X` 版本差异

- `2.X` 迭代方法名为 `X.next()` 而不是 `3.X` 的 `X.__next__()`。
- 出于可移植性，一个叫 `next(X)` 的内置函数出现在 `2.6` 及以后（包括 `3.X`）中，它用来调用 `3.X` 中的 `X.__next__()` 和 `2.X` 中的 `X.next()`。
- `2.6` 和 `2.7` 可以使用 `X.next()` 和 `next(X)`；`2.6` 之前需要用 `X.next()`，不能用 `next(X)`。

完整的迭代协议

- 可迭代对象 `iterable object`：迭代的被调对象，其 `__iter__` 方法被 `iter` 函数所调用
- 迭代器对象 `iterator object`：可迭代对象的返回结果，在迭代过程汇总实际提供值的对象。它的 `__next__` 方法被 `next` 运行，并在结束时触发 `StopIteration` 异常。

迭代器只支持一次迭代，而可迭代对象支持多次迭代（每次迭代重新调用 `iter`）。

> 这个说法有意思！

手动迭代

从技术上讲，`for` 循环内部的调用情况等价于 `I.__next__` 而不是 `next(I)`，尽管二者非常相似。

> 值得关注！

其他内置类型可迭代对象
在最新的 `Python` 版本中，字典作为一个可迭代对象自带一个迭代器，在迭代上下文中，会自动一次返回一个键：

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'b'
>>> next(I)
'c'
>>> next(I)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

`os.popen` 的返回结果也是可迭代的：

```
>>> import os
>>> P = os.popen('dir')
>>> P.__next__()
' Volume in drive C has no label.\n'
>>> P.__next__()
' Volume Serial Number is D093-D1F7\n'
>>> next(P)
TypeError: _wrap_close object is not an iterator
```

注意：在 `2.X` 中 `popen` 对象支持 `P.next()` 方法，`3.X` 中支持 `P.__next()` 方法，却不支持内置函数 `next(P)`。
由于 `next()` 被定义成调用 `__next__()`，因此这看上去有点不符合通常情况。
不过只要我们使用了被 `for` 循环以及其他迭代上下文自动采用的完整迭代协议，也就是顶层的 `iter` 调用，那么 `next()` 和 `__next__()` 就都能正常工作（`iter` 会执行一些内部必要的步骤，从而使该对象支持 `next` 调用）。

```
>>> P = os.popen('dir')
>>> I = iter(P)
>>> next(I)
' Volume in drive C has no label.\n'
>>> I.__next__()
' Volume Serial Number is D093-D1F7\n'
```

操作系统领域，`Python` 的标准路径遍历器 `os.walk` 是一个简单的可迭代对象。

`range` 返回的对象也是可迭代的

```
>>> R = range(5)
```

```
>>> R # Ranges are iterables in 3.X
range(0, 5)
>>> I = iter(R) # Use iteration protocol to produce results
>>> next(I)
0
>>> next(I)
1
>>> list(range(5)) # Or use list to collect all results at once
[0, 1, 2, 3, 4]
```

`enumerate` 工具也是可迭代的

```
>>> E = enumerate('spam') # enumerate is an iterable too
>>> E
<enumerate object at 0x00000000029B7678>
>>> I = iter(E)
>>> next(I) # Generate results with iteration protocol
(0, 's')
>>> next(I) # Or use list to force generation to run
(1, 'p')
>>> list(enumerate('spam'))
[(0, 's'), (1, 'p'), (2, 'a'), (3, 'm')]
```

14.2 列表推导：初次深入探索

遍历列表并修改（使用 `range` ）

```
>>> L = [1, 2, 3, 4, 5]
>>> for i in range(len(L)):
...     L[i] += 10
...
>>> L
[11, 12, 13, 14, 15]
```

使用列表推导：

```
>>> L = [x + 10 for x in L]
>>> L
[21, 22, 23, 24, 25]
```

列表推导的语法来源于集合理论表示法中的一个结构。
大多数人都会发现列表推导看上去就像是一个反着写的 `for` 循环。
取决于 `Python` 版本和代码，列表推导比手动的 `for` 循环语句运行的更快（往往速度会快一倍），这是因为它们的迭代在解释器内部是以 `C` 语言的速度执行的，而不是以手动 `Python` 代码

执行的。
尤其对于较大的数据集合，使用列表推导能带来极大的性能优势。

```
# 使用 for 循环手动构建列表
>>> res = []
>>> for x in L:
...     res.append(x + 10)
...
>>> res
[31, 32, 33, 34, 35]
```

每当我们需要在一个序列中的每项上执行一个操作时，就可以考虑使用列表推导。

> 这句话好好品味。

去除文件行尾空白：

```
>>> f = open('script2.py')
>>> lines = f.readlines()
>>> lines
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']

>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(x ** 32)']
```

简写形式：

```
>>> lines = [line.rstrip() for line in open('script2.py')]
>>> lines
['import sys', 'print(sys.path)', 'x = 2', 'print(x ** 32)']
```

其他示例：

```
>>> [line.upper() for line in open('script2.py')]
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']

>>> [line.rstrip().upper() for line in open('script2.py')]
['IMPORT SYS', 'PRINT(SYS.PATH)', 'X = 2', 'PRINT(X ** 32)']

>>> [line.split() for line in open('script2.py')]
[['import', 'sys'], ['print(sys.path)'], ['x', '=', '2'], ['print(x', '*
*', '32)']]

>>> [line.replace(' ', '!') for line in open('script2.py')]
['import!sys\n', 'print(sys.path)\n', 'x!=!2\n', 'print(x!**!32)\n']
```

```
>>> [('sys' in line, line[:5]) for line in open('script2.py')]
[(True, 'impor'), (True, 'print'), (False, 'x = 2'), (False, 'print')]
```

第二个示例：链式调用是有效的，因为字符串方法会返回一个新的字符串，可以对该字符串调用其他的字符串方法。

小细节：文件对象自身会在垃圾回收的时候自动关闭。列表推导也会自动地在表达式运行结束后，将它们的临时文件对象关闭。然而对于 `CPython` 以外的版本，可能需要手动关闭循环中的文件对象，以保证资源能够被立即释放。

扩展的列表推导语法
筛选分句：`if`

```
>>> lines = [line.rstrip() for line in open('script2.py') if line[0] ==
'p']
>>> lines
['print(sys.path)', 'print(x ** 32)']
```

等价的 `for` 循环形式：

```
>>> res = []
>>> for line in open('script2.py'):
...     if line[0] == 'p':
...         res.append(line.rstrip())
...
>>> res
['print(sys.path)', 'print(x ** 32)']
```

筛选出以数字结尾的行：

```
>>> [line.rstrip() for line in open('script2.py') if line.rstrip()[-1].is
digit()]
['x = 2']
```

排除空行的行数统计：

```
>>> fname = r'd:\books\5e\lp5e\draft1typos.txt'
>>> len(open(fname).readlines()) # All lines
263
>>> len([line for line in open(fname) if line.strip() != '']) # Nonblank
 lines
185
```

嵌套循环：`for`
列表推导式可以包含任意数目的 `for` 分句，并且每个 `for` 分句都带有可选的 `if` 子句。

有序组合 `ordered combination`

```
>>> [x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

等价 `for` 循环形式：

```
>>> res = []
>>> for x in 'abc':
...     for y in 'lmn':
...         res.append(x + y)
...
>>> res
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

14.3 其他迭代上下文

`iteration context`

```
>>> import functools, operator
>>> functools.reduce(operator.add, open('script2.py'))
'import sys\nprint(sys.path)\nx = 2\nprint(x ** 32)\n'
```

与 `map` 和其他函数不同，`sorted` 在 `3.X` 中会返回一个真正的列表而不是一个可迭代对象。

在当今的 `Python` 中，迭代协议甚至比我们目前已经展示的示例更为普遍——实际上 `Python` 的内置工具集中所有能够从左到右扫描一个对象的工具，都被定义为在主体对象上使用了迭代协议。这甚至包括了更高级的工具，例如 `list` 和 `tuple` 内置函数（它们从可迭代对象中构建一个新的对象），以及字符串 `join` 方法（它将一个子字符串放置到一个可迭代对象中包含的字符串之间，来创建一个新的字符串）。

```
>>> list(open('script2.py'))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']

>>> tuple(open('script2.py'))
('import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n')

>>> '&&'.join(open('script2.py'))
'import sys\n&&print(sys.path)\n&&x = 2\n&&print(x ** 32)\n'
```

> 个人总结：能够从左到右扫描一个对象的工具，它就应用了该对象的迭代协议，可以被认为是一个 *迭代工具*。

甚至其他的一些工具也出人意料地属于这个类别。例如，序列赋值、`in` 成员测试、切片赋值和列表的 `extend` 方法都利用了迭代协议来扫描，从而自动逐行读取文件：

```
>>> a, b, c, d = open('script2.py') # Sequence assignment
>>> a, d
('import sys\n', 'print(x ** 32)\n')

>>> a, *b = open('script2.py') # 3.X extended form
>>> a, b
('import sys\n', ['print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n'])

>>> 'y = 2\n' in open('script2.py') # Membership test
False
>>> 'x = 2\n' in open('script2.py')
True

>>> L = [11, 22, 33, 44] # Slice assignment
>>> L[1:3] = open('script2.py')
>>> L
[11, 'import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n',
44]

>>> L = [11]
>>> L.extend(open('script2.py')) # list.extend method
>>> L
[11, 'import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']
```

> 切片赋值的等号右边是可迭代对象，不用是列表。同样地，`extend` 接收的也是一个可迭代对象。

`append` 则不会自动迭代：

```
>>> L = [11]
>>> L.append(open('script2.py')) # list.append does not iterate
>>> L
[11, <_io.TextIOWrapper name='script2.py' mode='r' encoding='cp1252'>]
>>> list(L[1])
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x ** 32)\n']
```

`dict` 可以接收 `zip` 的返回结果。
`set` 也是个迭代工具：

```
>>> {line for line in open('script2.py') if line[0] == 'p'}
{'print(x ** 32)\n', 'print(sys.path)\n'}
>>> {ix: line for (ix, line) in enumerate(open('script2.py')) if line[0]
 == 'p'}
```

```
{1: 'print(sys.path)\n', 3: 'print(x ** 32)\n'}
```

推导表达式的亲戚：生成器表达式。

```
>>> list(line.upper() for line in open('script2.py'))  # See Chapter 20
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X ** 32)\n']
```

以下所有工具都接受任意的一个可迭代对象作为参数，并利用迭代协议来扫描它，但返回单个的结果：

```
>>> sum([3, 2, 4, 1, 5, 0])   # sum expects numbers only
15
>>> any(['spam', '', 'ni'])
True
>>> all(['spam', '', 'ni'])
False
>>> max([3, 2, 5, 1, 4])
5
>>> min([3, 2, 5, 1, 4])
1
>>> max(open('script2.py')) # Line with max/min string value
'x = 2\n'
>>> min(open('script2.py'))
'import sys\n'
```

`*参数`：后面的参数可以是任何可迭代对象，包括文件。

```
>>> def f(a, b, c, d): print(a, b, c, d, sep='&')
...
>>> f(1, 2, 3, 4)
1&2&3&4
>>> f(*[1, 2, 3, 4]) # Unpacks into arguments
1&2&3&4
>>>
>>> f(*open('script2.py')) # Iterates by lines too!
import sys
&print(sys.path)
&x = 2
&print(x ** 32)
```

`zip()` 相当于是转置矩阵

```
>>> X = (1, 2)
>>> Y = (3, 4)
>>>
>>> list(zip(X, Y))   # Zip tuples: returns an iterable
[(1, 3), (2, 4)]
```

```
>>>
>>> A, B = zip(*zip(X, Y)) # Unzip a zip!
>>> A
(1, 2)
>>> B
(3, 4)
```

`*zip()` 相当于是 `unzip a zip`，获得转置矩阵之后的每一行。

14.4 Python 3.X 新增的可迭代对象

`Python 3.X` 的一个本质改变是它比 `Python 2.X` 更强调迭代。
这一变化连同 `Unicode` 模型和强制新式类一起，是 `Python 3.X` 版本最彻底的变化。

好处：节约内存空间
缺点：影响编程风格（要用 `list` 一次性转换成列表）；只支持单次扫描（对于 `map` 和 `zip`）

```
>>> M = map(lambda x: 2 ** x, range(3))
>>> for i in M: print(i)
...
1
2
4
>>> for i in M: print(i) # Unlike 2.X lists, one pass only (zip too)
...
>>>
```

`range` 可迭代对象
取代了 `2.X` 的 `xrange`。
`3.X` 的 `range` 对象只支持迭代、索引以及 `len` 函数，不支持其他任何的序列操作。

```
C:\code> c:\python33\python
>>> R = range(10) # range returns an iterable, not a list
>>> R
range(0, 10)

>>> I = iter(R) # Make an iterator from the range iterable
>>> next(I) # Advance to next result
0 # What happens in for loops, comprehensions, etc.
>>> next(I)
1
>>> next(I)
2

>>> list(range(10)) # To force a list if required
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> len(R) # range also does len and indexing, but no others
10
>>> R[0]
0
>>> R[-1]
9
>>> next(I) # Continue taking from iterator, where left off
3
>>> I.__next__() # .next() becomes .__next__(), but use new next()
4
```

由于文件迭代器的出现，`2.X` 中用来最小化内存使用的 `file.xreadlines()` 方法也已经从 `3.X` 中移除了。

`map`、`zip` 和 `filter` 可迭代对象

与 `range` 不同，它们本身都是迭代器：在遍历其结果一次之后，它们就用尽了。

```
>>> M = map(abs, (-1, 0, 1)) # map returns an iterable, not a list
>>> M
<map object at 0x00000000029B75C0>
>>> next(M) # Use iterator manually: exhausts results
1 # These do not support len() or indexing
>>> next(M)
0
>>> next(M)
1
>>> next(M)
StopIteration

>>> for x in M: print(x) # map iterator is now empty: one pass only
...

>>> M = map(abs, (-1, 0, 1)) # Make a new iterable/iterator to scan again
>>> for x in M: print(x) # Iteration contexts auto call next()
...
1
0
1
>>> list(map(abs, (-1, 0, 1))) # Can force a real list if needed
[1, 0, 1]
```

> 本书中术语的区分：
>
> - 可迭代对象：实现了 `__iter__()` 方法，从而可以被 `iter()` 调用并返回一个迭代器的对象。
> - 迭代器：实现了 `__next__()` 方法，从而可以被 `next()` 调用，一次返回一个元素，最终抛出 `StopIteration` 异常的对象。

- 带有迭代器的可迭代对象：同时实现了 `__iter__()` 和 `__next__()` 方法；在 `__iter__()` 方法中返回自身 `return self`；在 `__next__()` 方法中一次返回一个元素，最终抛出 `StopIteration` 异常的对象。

```
>>> f = filter(bool, ['s', '', 1])
>>> for i in f:
...     print(i)
...
s
1
>>> list(f)
[]
```

多遍迭代器 vs 单遍迭代器

```
>>> Z = zip((1, 2, 3), (10, 11, 12))
>>> I1 = iter(Z)
>>> I2 = iter(Z) # Two iterators on one zip
>>> next(I1)
(1, 10)
>>> next(I1)
(2, 11)
>>> next(I2) # (3.X) I2 is at same spot as I1!
(3, 12)

>>> M = map(abs, (-1, 0, 1)) # Ditto for map (and filter)
>>> I1 = iter(M); I2 = iter(M)
>>> print(next(I1), next(I1), next(I1))
1 0 1
>>> next(I2) # (3.X) Single scan is exhausted!
StopIteration

>>> R = range(3) # But range allows many iterators
>>> I1, I2 = iter(R), iter(R)
>>> [next(I1), next(I1), next(I1)]
[0 1 2]
>>> next(I2) # Multiple active scans, like 2.X lists
0
```

使用类来编写自己的可迭代对象时，我们通常采用针对 `iter` 调用返回一个新的可迭代对象的方式，来支持多个迭代器；单个的迭代器一般意味着一个对象返回其自身。

生成器函数和表达式的行为就像 `map`、`zip` 和 `filter` 一样，支持单个迭代器。

字典视图可迭代对象

`3.X` 中，字典的 `keys`，`values` 和 `items` 方法返回可迭代的试图对象。
视图对象不是迭代器，但是是可迭代对象。

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'b': 2, 'c': 3}

>>> K = D.keys() # A view object in 3.X, not a list
>>> K
dict_keys(['a', 'b', 'c'])

>>> next(K) # Views are not iterators themselves
TypeError: dict_keys object is not an iterator

>>> I = iter(K) # View iterables have an iterator,
>>> next(I) # which can be used manually,
'a' # but does not support len(), index
>>> next(I)
'b'

>>> for k in D.keys(): print(k, end=' ') # All iteration contexts use aut
o
...
a b c
```

`3.X` 中字典本身是可迭代对象，同时带有一个返回连续的键的迭代器。

```
>>> D # Dictionaries still produce an iterator
{'a': 1, 'b': 2, 'c': 3} # Returns next key on each iteration
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'b'
>>> for key in D: print(key, end=' ') # Still no need to call keys() to i
terate
... # But keys is an iterable in 3.X too!
a b c
```

字典的键排序：

```
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> for k in sorted(D.keys()): print(k, D[k], end=' ')
...
a 1 b 2 c 3
>>> for k in sorted(D): print(k, D[k], end=' ') # "Best practice" key sor
ting
...
a 1 b 2 c 3
```

14.5 其他迭代话题

14.6 本章小结

1. `for` 循环和可迭代对象之间有什么关系？
   `for` 循环会使用迭代协议来遍历可迭代对象中的每一个项。
   它首先通过把可迭代对象传入 `iter` 函数从可迭代对象拿到一个迭代器。（对于一些自身就是迭代器的对象，用于初始化的 `iter` 调用时可有可无的。）
   然后再每次迭代中调用该迭代器对象的 `__next__` 方法，并捕捉 `StopIteration` 异常，从而决定何时停止循环。
2. `for` 循环和列表推导之间有什么关系？
   两者都是迭代工具和上下文。列表推导是执行常见 `for` 循环任务的简洁且高效的方法：对可迭代对象内所有的元素应用一个表达式，并收集其结果。
3. 举出 `Python` 中的 `4` 种迭代上下文 `iteration context`。
   - `for` 循环
   - 列表推导
   - `map`
   - `in`
   - `sorted` , `sum` , `any` , `all`
   - `list` 、 `tuple`
   - `join`
4. 目前逐行读取一个文本文件的最佳方式是什么？
   不要显式读取：在迭代上下文中打开文件（比如 `for` 循环或者列表推导），让迭代工具在每次迭代中执行该文件的 `next` 方法，自动一次扫描一行。
   从代码编写的简洁性、执行速度以及内存空间使用等方面来看，这种做法通常都是最佳的。

15. 文档

`PyDoc` 系统可以把模块内的文档，渲染成 `shell` 中的普通文本或是浏览器中的 `HTML` 页面。

15.1 Python 文档资源

Table 15-1. Python documentation sources

Form	Role
# comments	In-file documentation
The `dir` function	Lists of attributes available in objects
Docstrings: `__doc__`	In-file documentation attached to objects
PyDoc: the `help` function	Interactive help for objects
PyDoc: HTML reports	Module documentation in a browser
Sphinx third-party tool	Richer documentation for larger projects
The standard manual set	Official language and library descriptions
Web resources	Online tutorials, examples, and so on
Published books	Commercially polished reference texts

理解难度从上到下递减。
`dir` 函数：抓取对象内所有可用属性列表的一种简单方式。
如果不传入参数，它可以列出调用者作用域内的变量。
名称以双下划线开头的通常意味着与解释器相关。
名称以单下划线开头意味着私有属性。

```
>>> import sys
>>> len(dir(sys))
85
>>> len([x for x in dir(sys) if not x.startswith('__')])
74
>>> len([x for x in dir(sys) if not x[0] == '_'])
66
```

要查看列表和字符串的属性，可以传入空列表和空字符串。

```
>>> [i for i in dir(list) if not i.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>> [i for i in dir({}) if not i.startswith('__')]
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values']
```

包装到函数里面：

```
>>> def dir1(x): return [a for a in dir(x) if not a.startswith('__')] # S
ee Part IV
...
>>> dir1(tuple)
['count', 'index']
```

使用字面量和类型名称是等价的：

```
>>> dir(str) == dir('') # Same result, type name or literal
True
>>> dir(list) == dir([])
True
```

一些 `IDE` 拥有在图形界面上自动列出对象中属性的功能，从而可以看做是 `dir` 的替代。

文档字符串
`Python` 支持自动附加在对象上的文档，从而可以在运行时查看。
这种注释被写成字符串，放在模块文件、函数以及类语句的顶部，位于任何可执行代码之前（不过 `#` 注释以及 `UNIX` 风格的 `#!` 可以放在它们前面）
`Python` 会自动装在文档字符串的文本，使其成为相应对象的 `__doc__` 属性。

形式：

- 三重引号的多行块字符串
- 单行字符串

该文档协议的意义在于，你可以在文件被导入后，继续让注释保存在 `__doc__` 属性中以供查看。

一般你都需要使用 `print` 来打印文档字符串，否则你会得到一个嵌有换行符 `\n` 的字符串。

通过点号路径来访问 `module.class.method.__doc__`

```
>>> import sys
>>> print(sys.__doc__)

>>> print(sys.getrefcount.__doc__)
```

虽然你可以通过查看文档字符串来获取内置工具的大量信息，但其实你不必这样做：`help` 函数会为你自动完成这件事。

`PyDoc` 与 `help` 函数
标准的 `PyDoc` 工具是一段 `Python` 程序，用于提取文案的那个字符串及相关的结构化信息，并将它们排版成外观精美的多种报告。
两种最主要的 `PyDoc` 接口是内置的 `help` 函数同 `PyDoc` 基于 `GUI` 和基于 `Web` 的 `HTML` 报告接口。
类似 `UNIX` 系统的 `manpage`，使用回车到下一行，空格到下一页，`q` 键退出。

调用 `help` 函数不用导入 `sys`，但是如果想得到 `sys` 的信息必须导入。如果不导入，则要传入模块名称的字符串，比如 `import('sys.getrefcount')`（`3.3` 和 `2.7` 版本支持）

```
>>> help(sys.getrefcount)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sys' is not defined

>>> help('sys.getrefcount')
Help on built-in function getrefcount in sys:

sys.getrefcount = getrefcount(...)
    getrefcount(object) -> integer

    Return the reference count of object.  The count returned is generall
y
    one higher than you might expect, because it includes the (temporary)
    reference as an argument to getrefcount().
```

对诸如模块和类的大型对象而言，`help` 显示的内容会分解成多个部分。
其中一部分是 *文档字符串*，另一部分是 `PyDoc` 自动查看对象内部而收集的 *结构化信息*。

`help` 可以传入类型名称，或者该类型的实际对象，或者传入类型名称或一个十几对象的方法。
在最新的 `Python` 版本中直接在一个实际字符串对象上获取帮助信息是行不通的：因为字符串是被特殊解释的，会被当做请求一个未被导入的模块。所以必须使用 `str` 类型名。
还有一个交互帮助模式，交互模式下输入 `help()` 回车。

`PyDoc`：`HTML` 报告

- `3.3` 之前：`Python` 配有一个提交搜索请求的简易 `GUI` 桌面客户端。它会启动一个 `Web` 浏览器窗口，来查看通过自动运行的本地服务器生成的文档。
- `3.3` 开始：`GUI` 客户端被一个全浏览器接口方案所取代，该方案在一个网页中融合了搜索和显示，并且可以与一个自动运行的本地服务器进行通信。
- `3.2`：既支持原本的 `GUI` 客户端方案，也支持 `3.3` 版本中强制的新式全浏览器模式。

`Python 3.2` 及以前：
通过 “模块文档” `"Module Docs"` 开始按钮来启动 `GUI`，也可以通过命令行 `pydoc -g` 来启动。

`3.3` 中：
`pydoc -b` 启动。它既会启动一个本地运行的文档服务器，也会打开一个可作为搜索引擎客户端和进行页面显示的 `Web` 浏览器。

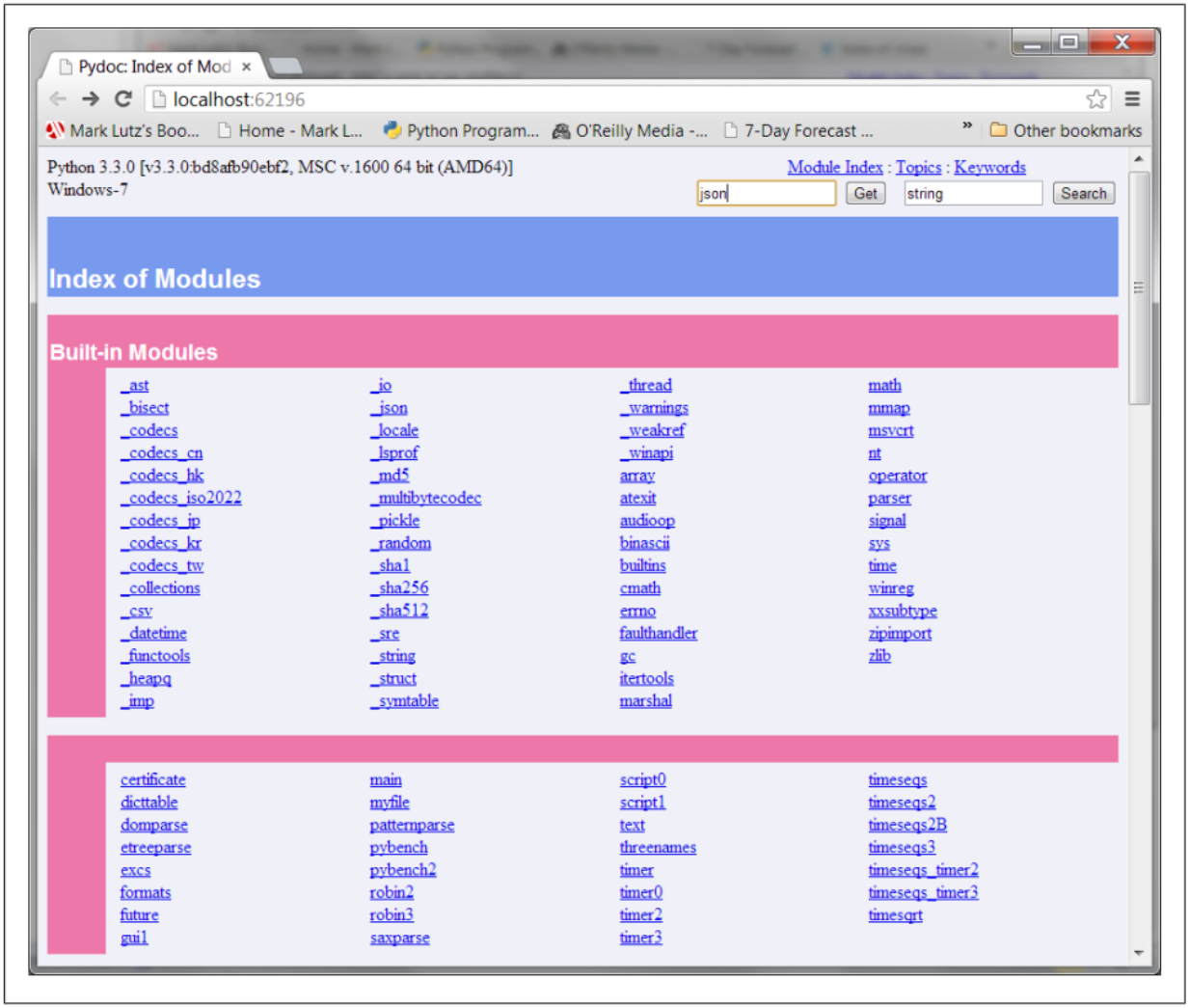
`-m` 这一 `Python` 命令行参数在模块导入路径搜索中搜索定位 `PyDoc` 的模块文件。
参阅附录 `A`

```
c:\code> python -m pydoc -b
Server ready at http://localhost:62135/
Server commands: [b]rowser, [q]uit
server> q
```

```
Server stopped

c:\code> py -3 -m pydoc -b
Server ready at http://localhost:62144/
Server commands: [b]rowser, [q]uit
server> q
Server stopped

c:\code> C:\python33\python -m pydoc -b
Server ready at http://localhost:62153/
Server commands: [b]rowser, [q]uit
server> q
Server stopped
```



启动 `PyDoc` 并让它作为在一个专有端口上的 `Web` 服务器，弹出一个 `Web` 浏览器窗口作为客户端，显示一个页面，其中包含了模块搜索路径（包括 `PyDoc` 被启动的目录）上所有可导入模块的文档的连接。
除模块索引以外，`PyDoc` 的网页也包括顶端的输入框，可允许你请求一个特定模块的文档页面 `Get` 并搜索相关的项 `Search`。

`pydoc -p port` 设置 `PyDoc` 服务器的端口
`pydoc -w module` 将模块的 `HTML` 文档写入一个名为 `module.html` 的文件，以便以后查看。

你也可以运行 `PyDoc` 来生成纯文本格式的文档，等效于交互模式的 `help` 调用：

```
c:\code> py -3 -m pydoc timeit # Command-line text help

c:\code> py -3
>>> help("timeit") # Interactive prompt text help
```

改变 `PyDoc` 的颜色
只能通过修改源代码来实现。
`Python` 标准库的 `pydoc.py` 文件。
`C:\Python33\Lib`
颜色是被硬编码的 `RGB` 值，`'#eeaa77'`。
作者用 `#008080` 青绿色代替 `#eeaa77` 暗粉色，用 `#c0c0c0` 灰色代替了 `#ffc8d8` 浅粉色。

`Python 3.2` 及更早：`GUI` 客户端

```
c:\code> c:\python32\python -m pydoc -g  # Explicit Python path
c:\code> py -3.2 -m pydoc -g             # Windows 3.3+ launcher version
```

`GUI` 界面既能用在内置模块上，也能用在用户定义的模块上。
你需要确保模块的外层目录位于模块导入搜索路径中。
因为 `PyDoc` 必须先导入一个文件才能渲染它的文档。
这也必须包括当前的工作目录：因为 `PyDoc` 可能不会检查它被启动的哪个目录。
所以你需要扩展 `PYTHONPATH` 设置来使其工作。
`3.2` 和 `2.7` 中，需要给 `PYTHONPATH` 加上一个 `.`，使 `PyDoc` 的 `GUI` 客户端模式能查询它通过命令行被启动的目录：

```
c:\code> set PYTHONPATH=.;%PYTYONPATH%
c:\code> py -3.2 -m pydoc -g
```

如果要在 `3.2` 中查看新的全浏览器 `pydoc -b` 模式下的的当前目录，这项设置也是必须的。
然而，`Python 3.3` 在它的索引列表中自动包含了 `.`，因此不用设置路径。

超越文档字符串：Sphinx
http://sphinx-doc.org
使用 `reStructuredText` 作为标记语言，并从能够解析和翻译 `reStructuredText` 的 `Docutils` 套件工具中继承了许多内容。

支持多种输出格式。
采用 `Pygments` 的自动代码高亮。

标准手册集
`Win 7` 及以前，既可以从开始菜单的 `Python` 子项中选取它，也可以从 `IDLE` 的 `Help` 选项菜单中打开它。
还可以单独从 http://www.python.org 官网下载不同格式的手册集。
或者在网站上在线阅读。

最重要的两项是：

- `Library Reference` 说明了内置类型、函数、异常以及标准库模块
- `Language Reference` 提供了官方的语言层面的细节说明

另外 `What's New` 按时间顺序记录了从 `2.0` 开始的每一个发行版所做的改变。

网络资源

已出版的书籍

15.2 常见代码编写陷阱

- 别忘了冒号
- 从第 `1` 列开始
- 空白行在交互式命令行下很重要
- 缩进要一致
- 不要在 `Python` 中编写 `C` 代码
- 使用简单 `for` 循环，而不是 `while` 或 `range`
- 注意赋值语句中的可变对象
  多目标赋值 `a = b = []`
  增量赋值 `a += [1, 2]`
  容易发生联动修改
- 不要期待在原位置修改对象的函数会返回结果
  `for k in D.keys().sort():`
  在 `2.X` 中，不会返回任何结果，变成了对 `None` 的循环
  在 `3.X` 中，`keys()` 得到的不是列表，没法 `sort()`
- 一定要使用括号来调用函数
- 不要在导入和重载中使用扩展名或路径
  `import mod` 而不是 `import mod.py`
  因为模块可能有除了 `.py` 以外的其他扩展名，比如 `.pyc`，所以硬编码一个特定的扩展名不仅是不合法的语法，而且也毫无意义。
  `Python` 会自动挑选一个扩展名。
- 其他部分的陷阱。

15.3 本章小结

15.4 本章习题

1. 什么时候应该使用文档字符串而不是 `#` 注释？
   文档字符串被认为是较大、功能性文档的最佳选择，用于描述程序中的模块、函数、类以及方法的使用。现在的 `#` 注释最好只限用于代码策略点中晦涩的表达式或语句的较小型文档。

一方面是因为文档字符串在源代码文件中比较容易找到，另一方面也是因为 `PyDoc` 系统能提取并显示。

2. 举出 `3` 种查看文档字符串的方式。
   - 打印对象的 `__doc__` 属性
   - 将对象传入 `help` 函数
   - `PyDoc` 基于 `HTML` 的用户界面
   - `3.2` 版本或更早的 `-g` ：`GUI` 客户端模式
   - `3.2` 版本或以后的 `-b` ：全浏览器模式
3. 如何获得对象中可用属性的列表？
   内置的 `dir(X)` 函数会返回附加在任何对象上的所有属性的列表。
4. 如何获得计算机中所有可用模块的列表？
   使用 `PyDoc`

15.5 第三部分练习题

`ord` 函数会返回 `Unicode` 编码，但如果把传入的字符限制在 `ASCII` 集中，那么你将得到 `ASCII` 编码。

字典键排序

```
>>> D = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
>>> D
{'f': 6, 'c': 3, 'a': 1, 'g': 7, 'e': 5, 'd': 4, 'b': 2}
>>>
>>> keys = list(D.keys()) # list() required in 3.X, not in 2.X
>>> keys.sort()
>>> for key in keys:
...     print(key, '=>', D[key])
...
a => 1
b => 2
c => 3
d => 4
e => 5
f => 6
g => 7

>>> for key in sorted(D): # Better, in more recent Pythons
...     print(key, '=>', D[key])
```

`Win` 文件比较 `fc`

完成于 `2019.1.2`