Python中文手册

1 数字

除法'/'永远返回一个浮点数，如果要得到整数结果，可以使用'//'运算符，要计算余数可以使用'

可以使用'**'计算幂乘方

等号'='用于给变量赋值，赋值之后，在下一个提示符之前不会有任何结果显示。

整数和浮点数的混合计算中，整数会被转换成浮点数

Python还支持复数，使用后缀j或J表示虚数部分。

2 字符串

输出的字符串会用引号引起来，特殊字符会用反斜杠转义。如果字符中只有单引号没有双引号，就用双引号引用，否则用单引号引用。

print()函数生成可读性更好的输出，他会省去引号并且打印出转以后的特殊字符。

如果你前面带有反斜杠的字符被当做特殊字符，你可以使用原始字符串，方法是在第一个引号前面加上一个'r'。

字符转文本可以分成多行。一种方法是引用三引号："""..."""或者"'..."'。行尾换行符会被自动包含到字符串中，但是可以在行位加上反斜杠来避免这个行为。

字符串可以用'+'操作符连接，可以由'*'表示重复

相邻的两个字符串文本自动连接在一起

它只用于两个字符串文本，不能用于字符串表达式

如果你想连接多个变量或者连接一个变量或一个字符串文本，使用'+'

字符串也可以被检索，字符串的第一个字符索引为0。 索引也可以为负数，表示从右边开始计算。

索引用于获得单个字符，切片让你获得一个子字符串

注意：包含起始的字符，不包含末尾的字符

切片的索引有非常有用的默认值，省略的第一个索引默认为零，省略的第二个索引默认为切片的字符串的大小。

对于非负索引，如果上下都在边界内，切片长度就是两个索引之差。

Python字符串不可以被更改——它们是不可变的。因此，赋值给字符串索引的位置会导致错误

如果你需要一个不同的字符串，你应该创建一个新的

内置函数len()返回字符串长度

3 列表

Python有几个复合数据类型，用于表示其它的值。最通用的是list(列表)，他可以写作中括号之间的一列逗号分隔的值。

列表的元素不必是同一类型

就像字符串（以及其他所有内建的序列类型一样，列表可以被索引和切片

所有的切片操作都会返回一个包含请求的元素的新列表。

列表也支持连接的操作。

不像不可变的字符串，列表是可变的，它允许修改元素

你还可以使用append()方法在列表的末尾添加新的元素。也可以对切片赋值，此操作可以改变列表的尺寸，或清空它。

内置函数len()同样适用于列表

允许嵌套列表（创建一个包含其他列表的列表）

4 编程的第一步

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

(1)第一行包括了一个 多重赋值：变量 a 和 b 同时获得了新的值 0 和 1 最后一行又使用了一次。

在这个演示中，变量赋值前，右边首先完成计算。右边的表达式从左到右计算。

(2)条件（这里是 b ¡ 10 ）为 true 时， while 循环执行。在 Python 中，类似于 C，任何非零整数都是 true；0 是 false。条件也可以是字符串或列表，实际上可以是任何序列；

所有长度不为零的是 true，空序列是 false。示例中的测试是一个简单的比较。标准比较操作符与 C 相同： ¡ ， ¿ ， == ， ¡=， ¿= 和 !=。

(3)循环 体 是 缩进 的：缩进是 Python 组织语句的方法。Python (还)不提供集成的行编辑功能，所以你要为每一个缩进行输入 TAB 或空格。

(4)关键字 print() 语句输出给定表达式的值。它控制多个表达式和字符串输出为你想要字符串（就像我们在前面计算器的例子中那样）。

字符串打印时不用引号包围，每两个子项之间插入空间，所以你可以把格式弄得很漂亮，像这样:

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

用一个逗号结尾就可以禁止输出换行:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=',')
...     a, b = b, a+b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

5 if语句

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print('Negative changed to zero')
... elif x == 0:
...      print('Zero')
... elif x == 1:
...      print('Single')
... else:
...      print('More')
...
More
```

可能会有零到多个 elif 部分，else 是可选的。关键字 'elif' 是 'else if' 的缩写，这个可以有效地避免过深的缩进。if ... elif ... elif ... 序列用于替代其它语言中的 switch 或 case 语句。

6 for语句

Python 的 for 语句依据任意序列（链表或字符串）中的子项，按它们在序列中的顺序来进行迭代。例如（没有暗指）：

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

minted python 在迭代过程中修改迭代序列不安全（只有在使用链表这样的可变序列时才会有这样的情况）。如果你想要修改你迭代的序列（例如，复制选择项），你可以迭代它的复本。使用切割标识就可以很方便的做到这一点:

```
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

7 range()函数

如果你需要一个数值序列，内置函数 range() 会很方便，它生成一个等差级数链表:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

range(10) 生成了一个包含 10 个值的链表，它用链表的索引值填充了这个长度为 10 的列表，所生成的链表中不包括范围中的结束值。也可以让 range() 操作从另一个数值开始，或者可以指定一个不同的步进值（甚至是负数，有时这也被称为“步长”）:

```
range(5, 10)
   5 through 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

需要迭代链表索引的话，如下所示结合使 用 range() 和 len()

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
```

```
1 had
2 a
3 little
4 lamb
```

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

8 break 和 continue 语句, 以及循环中的 else 子句

循环可以有一个 else 子句；它在循环迭代完整个列表（对于 for ）或执行条件为 false （对于 while ）时执行，但循环被 break 中止的情况下不会执行。以下搜索素数的示例程序演示了这个子句:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, 这是正确的代码。看仔细：else 语句是属于 for 循环之中， 不是 if 语句。)

continue 语句是从 C 中借鉴来的，它表示循环继续执行下一次迭代:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
```

```
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

9 pass 语句

pass 语句什么也不做。它用于那些语法上必须要有什么语句，但程序什么也不做的场合，例如:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

10 定义函数

我们可以创建一个用来生成指定边界的斐波那契数列的函数:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

关键字 def 引入了一个函数 定义。在其后必须跟有函数名和包括形式参数的圆括号。函数体语句从下一行开始，必须是缩进的。

函数 调用 会为函数局部变量生成一个新的符号表。确切的说，所有函数中的变量赋值都是将值存储在局部符号表。变量引用首先在局部符号表中查找，然后是包含函数的局部符号表，然后是全局符号表，最后是内置名字表。因此，全局变量不能在函数中直接赋值（除非用 global 语句命名），尽管他们可以被引用。

函数引用的实际参数在函数调用时引入局部符号表，因此，实参总是 传值调用 （这里的 值 总是一个对象 引用 ，而不是该对象的值）。一个函数被另一个函数调用时，一个新的局部符号表在调用过程中被创建。

一个函数定义会在当前符号表内引入函数名。函数名指代的值（即函数体）有一个被 Python 解释器认定为 用户自定义函数 的类型。 这个值可以赋予其他的名字（即变量名），然后它也可以被当做函数使用。这可以作为通用的重命名机制:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

定义一个返回斐波那契数列数字列表的函数，而不是打印它，是很简单的:

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

和以前一样，这个例子演示了一些新的 Python 功能:

(1)return 语句从函数中返回一个值，不带表达式的 return 返回 None。过程结束后也会返回 None。

(2)语句 result.append(b) 称为链表对象 result 的一个 方法。方法是一个“属于”某个对象的函数，它被命名为 obj.methodename，这里的 obj 是某个对象（可能是一个表达式）， methodename 是某个在该对象类型定义中的方法的命名。

(3)示例中演示的 append() 方法由链表对象定义，它向链表中加入一个新元素。在示例中它等同于 result = result + [a]，不过效率更高。

11 默认参数值

最常用的一种形式是为一个或多个参数指定默认值。这会创建一个可以使用比定义时允许的参数更少的参数调用的函数，例如:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise OSError('uncooperative user')
        print(complaint)

#(1)只给出必要的参数:

ask_ok('Do you really want to quit?')

#(2)给出一个可选的参数:

ask_ok('OK to overwrite the file?', 2)

#(3)或者给出所有的参数:

ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

重要警告: 默认值只被赋值一次。这使得当默认值是可变对象时会有所不同，比如列表、字典或者大多数类的实例。例如，下面的函数在后续调用过程中会累积（前面）传给它的参数:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

这将输出:

```
[1]
[1, 2]
[1, 2, 3]
```

如果你不想让默认值在后续调用中累积，你可以像下面一样定义函数:

```
def f(a, L=None):
    if L is None:
```

```
        L = []
    L.append(a)
    return L
```

12 关键字参数

函数可以通过 关键字参数 的形式来调用，形如 keyword = value。例如，以下的函数:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

接受一个必选参数 (voltage) 以及三个可选参数 (state, action, 和 type)。可以用以下的任一方法调用:

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

不过以下几种调用是无效的:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

在函数调用中，关键字的参数必须跟随在位置参数的后面。传递的所有关键字参数必须与函数接受的某个参数相匹配 （例如 actor 不是 parrot 函数的有效参数），它们的顺序并不重要。这也包括非可选参数（例如 parrot(voltage=1000) 也是有效的）。任何参数都不可以多次赋值。下面的示例由于这种限制将失败:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

引入一个形如 **name 的参数时，它接收一个字典（参见 Mapping Types — dict ），该字典包含了所有未出现在形式参数列表中的关键字参数。这里可能还会组合使用一个形如 *name （下一小节详细介绍） 的形式参数，它接收一个元组（下一节中会详细介绍），包含了所有没有出现在形式参数列表中的参数值（ *name 必须在 **name 之前出现）。例如，我们这样定义一个函数:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    keys = sorted(keywords.keys())
    for kw in keys:
        print(kw, ":", keywords[kw])
#它可以这样被调用
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
#当然他会按如下内容打印:
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

注意在打印关键字参数之前，通过对关键字字典 keys() 方法的结果进行排序，生成了关键字参数名的列表；如果不这样做，打印出来的参数的顺序是未定义的。

13 可变参数列表

最后，一个最不常用的选择是可以让函数调用可变个数的参数。这些参数被包装进一个元组（参见 元组和序列 ）。在这些可变个数的参数之前，可以有零到多个普通的参数:

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

通常，这些 可变 参数是参数列表中的最后一个，因为它们将把所有的剩余输入参数传递给函数。任何出现在 *args 后的参数是关键字参数，这意味着，

他们只能被用作关键字，而不是位置参数:

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

14 参数列表的分拆

另有一种相反的情况: 当你要传递的参数已经是一个列表，但要调用的函数却接受分开一个个的参数值。这时候你要把已有的列表拆开来。例如内建函数 range() 需要要独立的 start，stop 参数。你可以在调用函数时加一个 * 操作符来自动把参数列表拆开:

```
>>> list(range(3, 6))            # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # call with arguments unpacked from a list
[3, 4, 5]
\paragraph{}
以同样的方式，可以使用 ** 操作符分拆关键字参数为字典:
\begin{minted}{python}
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised
```

15 Lambda形式

通过 lambda 关键字，可以创建短小的匿名函数。这里有一个函数返回它的两个参数的和: lambda a, b: a+b。 Lambda 形式可以用于任何需要的函数对象。出于语法限制，它们只能有一个单独的表达式。语义上讲，它们只是普通函数定义中的一个语法技巧。类似于嵌套函数定义，lambda 形式可以从外部作用域引用变量:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
```

```
>>> f(0)
42
>>> f(1)
43
```

上面的示例使用 lambda 表达式返回一个函数。另一个用途是将一个小函数作为参数传递

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

16 文档字符串

以下是一个多行文档字符串的示例

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

17 函数注解

```
#注解是以字典形式存储在函数的 __annotations__ 属性中，对函数的其它部分没有任何影响。
#参数注解（Parameter annotations）是定义在参数名称的冒号后面，紧随着一个表示注解的值得表达式。
#返回注释（Return annotations）是定义在一个 -> 后面，紧随着一个表达式，在冒号与 -> 之间。
#下面的示例包含一个位置参数，一个关键字参数，和没有意义的返回值注释：

>>> def f(ham: 42, eggs: int = 'spam') -> "Nothing to see here":
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...
>>> f('wonderful')
Annotations: {'eggs': <class 'int'>, 'return': 'Nothing to see here', 'ham': 42}
Arguments: wonderful spam
```

18 编码风格

(1)把空格放到操作符两边，以及逗号后面，但是括号里侧不加空格

19 关于列表

```
list.append(x)
#把一个元素添加到列表的结尾，相当于 a[len(a):] = [x]
\paragraph{}
list.extend(L)
#将一个给定列表中的所有元素都添加到另一个列表中，相当于 a[len(a):] = L
\paragraph{}
list.insert(i, x)
#在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，
#例如 a.insert(0, x) 会插入到整个列表之前，而 a.insert(len(a), x) 相当于 a.append(x)。
\paragraph{}
list.remove(x)
#删除列表中值为 x 的第一个元素。如果没有这样的元素，就会返回一个错误。
\paragraph{}
list.pop([i])
#从列表的指定位置删除元素，并将其返回。如果没有指定索引，a.pop() 返回最后一个元素。
#元素随即从列表中被删除（方法中i两边的方括号表示这个参数是可选的，而不是要求你输入一对方括号）
\paragraph{}
list.clear()
#从列表中删除所有元素。相当于 del a[:]。
\paragraph{}
list.index(x)
#返回列表中第一个值为 x 的元素的索引。如果没有匹配的元素就会返回一个错误。
\paragraph{}
list.count(x)
#返回 x 在列表中出现的次数。
\paragraph{}
list.sort()
#对列表中的元素就地进行排序。
\paragraph{}
list.reverse()
#就地倒排列表中的元素。
\paragraph{}
list.copy()
#返回列表的一个浅拷贝。等同于 a[:]。
```

下面这个示例演示了列表的大部分方法:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
#也许大家会发现像 insert, remove 或者 sort 这些修改列表的方法没有打印返回值,它们返回 None。
```

20 把列表当作堆栈使用

用 append() 方法可以把一个元素添加到堆栈顶。用不指定索引的 pop() 方法可以把一个元素从堆栈顶释放出来。例如:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

21 把列表当作队列使用

要实现队列，使用 collections.deque，它为在首尾两端快速插入和删除而设计。例如:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

22 列表推倒式

列表推导式为从序列中创建列表提供了一个简单的方法。普通的应用程式通过将一些操作应用于序列的每个成员并通过返回的元素创建列表，或者通过满足特定条件的元素创建子序列。

例如, 假设我们创建一个 squares 列表, 可以像下面方式:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

注意这个 for 循环中的被创建(或被重写)的名为 x 的变量在循环完毕后依然存在。使用如下方法，我们可以计算squares的值而不会产生任何的副作用:

```
squares = list(map(lambda x: x**2, range(10)))
```

或者，等价于:

```
squares = [x**2 for x in range(10)]
```

列表推导式由包含一个表达式的括号组成，表达式后面跟随一个 for 子句，之后可以有零或多个 for 或 if 子句。结果是一个列表，由表达式依据其后面的 for 和 if 子句上下文计算而来的结果构成。

例如，如下的列表推导式结合两个列表的元素，如果元素之间不相等的话:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

等同于:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

如果想要得到一个元组（例如，上面例子中的 (x, y)），必须要加上括号:

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in ?
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

列表推导式可使用复杂的表达式和嵌套函数:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

23 嵌套的列表推导式

考虑下面由三个长度为 4 的列表组成的 3x4 矩阵:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

现在，如果你想交换行和列，可以用嵌套的列表推导式:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

像前面看到的，嵌套的列表推导式是对 for 后面的内容进行求值，所以上例就等价于:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

反过来说，如下也是一样的:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

在实际中，你应该更喜欢使用内置函数组成复杂流程语句。对此种情况 zip() 函数将会做的更好:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

24 del语句

有个方法可以从列表中按给定的索引而不是值来删除一个子项： del 语句。它不同于有返回值的 pop() 方法。语句 del 还可以从列表中删除切片或清空整个列表（我们以前介绍过一个方法是将空列表赋值给列表的切片）。例如:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

del 也可以删除整个变量:

```
>>> del a
```

25 元组和序列

我们知道列表和字符串有很多通用的属性，例如索引和切割操作。它们是 序列 类型（参见 Sequence Types — list, tuple, range ）中的两种。因为 Python 是一个在不停进化的语言，也可能会加入其它的序列类型，这里介绍另一种标准序列类型： 元组 。

一个元组由数个逗号分隔的值组成，例如:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
```

```
>>> v
([1, 2, 3], [3, 2, 1])
```

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可以有或没有括号，不过经常括号都是必须的（如果元组是一个更大的表达式的一部分）。不能给元组的一个独立的元素赋值（尽管你可以通过联接和切割来模拟）。还可以创建包含可变对象的元组，例如列表。

虽然元组和列表很类似，它们经常被用来在不同的情况和不同的用途。元组有很多用途。例如 (x, y) 坐标对，数据库中的员工记录等等。元组就像字符串， 不可变的。通常包含不同种类的元素并通过分拆（参阅本节后面的内容）或索引访问（如果是 namedtuples，甚至可以通过属性）。列表是 可变的 ，它们的元素通常是相同类型的并通过迭代访问。

一个特殊的问题是构造包含零个或一个元素的元组：为了适应这种情况，语法上有一些额外的改变。一对空的括号可以创建空元组；要创建一个单元素元组可以在值后面跟一个逗号（在括号中放入一个单值不够明确）。丑陋，但是有效。例如:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

26 集合

Python 还包含了一个数据类型 —— set （集合）。集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。集合对象还支持union(联合),intersection(交),difference(差)和sysmmetric difference(对称差集)等数学运算。

```
#大括号或 set() 函数可以用来创建集合。
#注意：想要创建空集合，你必须使用 set() 而不是 {}。#后者用于创建空字典，我们在下一节中介绍的一种
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
```

```
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

类似 列表推导式，这里有一种集合推导式语法:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

27 字典

序列是以连续的整数为索引，与此不同的是，字典以 关键字 为索引，关键字可以是任意不可变类型，通常用字符串或数值。如果元组中只包含字符串和数字，它可以做为关键字，如果它直接或间接的包含了可变对象，就不能当做关键字。不能用列表做关键字，因为列表可以用索引、切割或者 append() 和 extend() 等方法改变。

理解字典的最佳方式是把它看做无序的键: 值对 (key:value 对) 集合，键必须是互不相同的（在同一个字典之内）。一对大括号创建一个空的字典: 。初始化列表时，在大括号内放置一组逗号分隔的键: 值对，这也是字典输出的方式。

字典的主要操作是依据键来存储和析取值。也可以用 del 来删除键: 值对 (key:value)。如果你用一个已经存在的关键字存储值，以前为该关键字分配的值就会被遗忘。试图从一个不存在的键中取值会导致错误。

```
#对一个字典执行 list(d.keys()) 将返回一个字典中所有关键字组成的无序列表（如果你想要排序，只需使
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
```

```
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

dict() 构造函数可以直接从 key-value 对中创建字典:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

此外，字典推导式可以从任意的键值表达式中创建字典:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

如果关键字都是简单的字符串，有时通过关键字参数指定 key-value 对更为方便:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

28 循环技巧

在字典中循环时，关键字和对应的值可以使用 items() 方法同时解读出来:

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

在序列中循环时，索引位置和对应值可以使用 enumerate() 函数同时得到:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

同时循环两个或更多的序列，可以使用 zip() 整体打包:

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

需要逆向循环序列的话，先正向定位序列，然后调用 reversed() 函数:

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

要按排序后的顺序循环序列的话，使用 sorted() 函数，它不改动原序列，而是生成一个新的已排序的序列:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

若要在循环内部修改正在遍历的序列（例如复制某些元素），建议您首先制作副本。在序列上循环不会隐式地创建副本。切片表示法使这尤其方便:

```
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words[:]:  # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

29 深入条件控制

while 和 if 语句中使用的条件不仅可以使用比较，而且可以包含任意的操作。

比较操作符 in 和 not in 审核值是否在一个区间之内。操作符 is 和 is not 比较两个对象是否相同；这只和诸如列表这样的可变对象有关。所有的比较操作符具有相同的优先级，低于所有的数值操作。

比较操作可以传递。例如 a ¡ b == c 审核是否 a 小于 b 并且 b 等于 c。

比较操作可以通过逻辑操作符 and 和 or 组合，比较的结果可以用 not 来取反义。这些操作符的优先级又低于比较操作符，在它们之中，not 具有最高的优先级， or 优先级最低，所以 A and not B or C 等于 (A and (notB)) or C。当然，括号也可以用于比较表达式。

逻辑操作符 and 和 or 也称作短路操作符：它们的参数从左向右解析，一旦结果可以确定就停止。例如，如果 A 和 C 为真而 B 为假， A and B and C 不会解析 C。作用于一个普通的非逻辑值时，短路操作符的返回值通常是最后一个变量。

可以把比较或其它逻辑表达式的返回值赋给一个变量，例如:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

30 比较序列和其他类型

序列对象可以与相同类型的其它对象比较。比较操作按 字典序 进行：首先比较前两个元素，如果不同，就决定了比较的结果；如果相同，就比较后两个元素，依此类推，直到所有序列都完成比较。如果两个元素本身就是同样类 型的序列，就递归字典序比较。如果两个序列的所有子项都相等，就认为序列相等。如果一个序列是另一个序列的初始子序列，较短的一个序列就小于另一个。字符 串的字典序按照单字符的 ASCII 顺序。下面是同类型序列之间比较的一些例子:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

需要注意的是如果通过 ¡ 或者 ¿ 比较的对象只要具有合适的比较方法就是合法的。比如，混合数值类型是通过它们的数值进行比较的，所以 0 是等于 0.0 。否则解释器将会触发一个 TypeError 异常，而不是提供一个随意的结果。

31 模块

```
#模块是包括 Python 定义和声明的文件。文件名就是模块名加上 .py 后缀。
#模块的模块名（做为一个字符串）可以由全局变量 __name__ 得到。
#例如，你可以用自己惯用的文件编辑器在当前目录下创建一个叫 fibo.py 的文件，录入如下内容：

# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

现在进入 Python 解释器并使用以下命令导入这个模块:

```
>>> import fibo
#这样做不会直接把 fibo 中的函数导入当前的语义表；它只是引入了模块名 fibo。你可以通过模块名按如下
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
```

```
'fibo'
#如果打算频繁使用一个函数，你可以将它赋予一个本地变量：
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

32 输入和输出

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

有两种方式可以写平方和立方表

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
```

```
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

str.zfill() 它用于向数值的字符串表达左侧填充 0。该函数可以正确理解正负号:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

方法 str.format() 的基本用法如下:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

大括号和其中的字符会被替换成传入 str.format() 的参数。大括号中的数值指明使用传入 str.format() 方法的对象中的哪一个:

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

如果在 str.format() 调用时使用关键字参数，可以通过参数名来引用值:

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

位置参数和关键字参数可以随意组合:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
                                                       other='Georg'))
The story of Bill, Manfred, and Georg.
```

```
#'!a' (应用 ascii()), '!s' (应用 str() ) 和 '!r' (应用 repr() ) 可以在格式化之前转换值:
>>> import math
>>> print('The value of PI is approximately {}.'.format(math.pi))
The value of PI is approximately 3.14159265359.
>>> print('The value of PI is approximately {!r}.'.format(math.pi))
The value of PI is approximately 3.141592653589793.
```

在字段后的 ':' 后面加一个整数会限定该字段的最小宽度，这在美化表格时很有用:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, phone))
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

33 文件读写

```
#函数 open() 返回 文件对象，通常的用法需要两个参数: open(filename, mode)。
>>> f = open('workfile', 'w')
```

第一个参数是一个含有文件名的字符串。第二个参数也是一个字符串，含有描述如何使用该文件的几个字符。mode 为 'r' 时表示只是读取文件；'w' 表示只是写入文件（已经存在的同名文件将被删掉）；'a' 表示打开文件进行追加，写入到文件中的任何数据将自动添加到末尾。 'r+' 表示打开文件进行读取和写入。mode 参数是可选的，默认为 'r'。