Python

第一章 数据类型和变量

1 变量名和关键字

变量是命名位置，用于存储对内存中存储的对象的引用。 我们为变量和函数选择的名称通常称为 `标识符` 。 在 Python 中，标识符必须遵守以下规则。

1. 所有标识符都必须以字母或下划线（ `_` ）开头，您不能使用数字。 例如： `my_var` 是有效的标识符，但 `1digit` 不是。
2. 标识符可以包含字母，数字和下划线（ `_` ）。 例如： `error_404` ， `_save` 是有效的标识符，但 `$name$` （不允许 `$` ）和 `#age` （不允许 `#` ）是无效的标识符。
3. 它们可以是任何长度。
4. 标识符不能是关键字。 关键字是 Python 用于特殊目的的保留字）。 以下是 Python 3 中的关键字。

```
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
pass       else       import     assert
break      except     in         raise
```

2 变量赋值

> - 变量和值是一对关系。变量赋值，是把数据的引用或者说地址，传递给某个变量，进行标记。所有的变量都是引用。指向具体的地址。
> - 引用和指针不同。引用是同一个数据的别名。指针是一个数据类型，存储的是指向数据的地址。引用不是一个数据类型。是一种变量形式。
> - 数值类型与字符串类型的值是不可变的。一旦定义就不能更改，内存中会分配固定大小的位置。更改会创建一个新的值。
> - 列表、元组、字典等复合类型的值是可变的。可以通过下表索引来操作内部元素的值。其地址不会变化，更改不会创建新的复合类型。
> - 自定义类型（class）的值也是可变的。在同一个地址的一个对象，可以更改对象内部的数据。

值是程序可以使用的基本东西。 例如： `1` ， `11` ， `3.14` 和 `"hello"` 均为值。 在编程术语中，它们通常也称为字面值。 字面值可以是不同的类型，例如 `1` ， `11` 是 `int` 类型， `3.14` 是 `float` 和 `"hello"` 是 `string` 。 记住，在 Python 中，所有东西都是对象，甚至是基本数据类型，例如 int，float，string，我们将在后面的章节中对此进行详细说明。

在 Python 中，您不需要提前声明变量类型。 解释器通过包含的数据自动检测变量的类型。 要将值赋给变量等号（ `=` ）。 `=` 符号也称为赋值运算符。

以下是变量声明的一些示例：

```
x = 100                                 # x is integer
pi = 3.14                               # pi is float
empname = "python is great"    # empname is string

a = b = c = 100 # this statement assign 100 to c, b and a.

```

提示：

将值分配给变量后，该变量本身不存储值。 而是，变量仅将对象的引用（地址）存储在内存中。 因此，在上面的清单中，变量 `x` 存储对 `100` （ `int` 对象）的引用（或地址）。 变量 `x` 本身不存储对象 `100` 。

3 注释

注释是说明程序目的或程序工作方式的注释。 注释不是 Python 解释器在运行程序时执行的编程语句。 注释也用于编写程序文档。 在 Python 中，任何以井号（ `#` ）开头的行均被视为注释。 例如：

```
# This program prints "hello world"
print("hello world")
```

在此清单中，第 1 行是注释。 因此，在运行程序时，Python 解释器将忽略它。

我们还可以在语句末尾写注释。 例如：

```
# This program prints "hello world"

print("hello world")   # display "hello world"

```

当注释以这种形式出现时，它们称为 `最终注释` 。

4 多重赋值

同时赋值或多重赋值允许我们一次将值赋给多个变量。 同时分配的语法如下：

```
var1, var2, ..., varn = exp1, exp2, ..., expn
```

该语句告诉 Python 求值右边的所有表达式，并将它们分配给左边的相应变量。 例如：

```
a, b = 10, 20

print(a)
print(b)
```

当想交换两个变量的值时，同时分配非常有帮助。 例如：

```
>>> x = 1    # initial value of x is 1
>>> y = 2    # initial value of y is 2

>>> y, x = x, y # assign y value to x and x value to y

```

预期输出：

```
>>> print(x)   # final value of x is 2
2
>>> print(y)   # final value of y is 1
1
```

5 Python 数据类型

Python 即有 5 种标准数据类型。

1. 数值
2. 字符串
3. 列表
4. 元组
5. 字典
6. 布尔值 - 在 Python 中， `True` 和 `False` 是布尔字面值。 但是以下值也被认为是 `False` 。
   - `0` - `0` ， `0.0`
   - `[]` - 空列表， `()` -空元组， `{}` -空字典， `''`
   - `None`

6 从控制台接收输入

`input()` 函数用于从控制台接收输入。

语法： `input([prompt]) -> string`

`input()` 函数接受一个名为 `prompt` 的可选字符串参数，并返回一个字符串。

```
>>> name = input("Enter your name: ")
>>> Enter your name: tim
>>> name
'tim'

```

请注意，即使输入了数字， `input()` 函数也始终返回字符串。 要将其转换为整数，可以使用 `int()` 或 `eval()` 函数。

```
>>> age = int(input("Enter your age: "))
Enter your age: 22
>>> age
22
>>> type(age)
<class 'int'>
```

7 导入模块

Python 使用模块组织代码。 Python 随附了许多内置模块，可用于例如数学相关函数的 `math` 模块，正则表达式的 `re` 模块，与操作系统相关的函数的 `os` 模块等。

要使用模块，我们首先使用 `import` 语句将其导入。 其语法如下：

```
import module_name
```

我们还可以使用以下语法导入多个模块：

```
import module_name_1, module_name_2
```

这是一个例子

```
>>> import math, os
>>>
>>> math.pi
3.141592653589793
>>>
>>> math.e
2.718281828459045
>>>
>>>
>>> os.getcwd()   # print current working directory
>>> '/home/user'
>>>
```

在此清单中，第一行导入了 `math` 和 `os` 模块中定义的所有函数，类，变量和常量。 要访问模块中定义的对象，我们首先编写模块名称，后跟点（`.`），然后编写对象本身的名称。 （即类或函数或常量或变量）。 在上面的示例中，我们从 `math` 数学访问两个常见的数学常数 `pi` 和 `e`。 在下一行中，我们将调用 `os` 模块的 `getcwd()` 函数，该函数将打印当前工作目录。

第二章 数字和运算

此数据类型仅支持诸如 `1`， `31.4`， `-1000`， `0.000023` 和 `88888888` 之类的数值。

Python 支持 3 种不同的数字类型。

1. `int` -用于整数值，例如 `1`， `100`， `2255`， `-999999`， `0` 和 `12345678`。
2. `float` -用于像 `2.3`， `3.14`， `2.71`， `-11.0` 之类的浮点值。
3. `complex` -适用于 `3+2j`， `-2+2.3j`， `10j`， `4.5+3.14j` 等复数。

1 整数

python 中的整数字面值属于 `int` 类。

```
>>> i = 100
>>> i
100
```

2 浮点数

浮点数是带有小数点的值。

```
>>> f = 12.3
>>> f
12.3
```

需要注意的一点是，当数字运算符的操作数之一是浮点值时，结果将是浮点值。

```
>>> 3 * 1.5
4.5
```

3 复数

如您所知，复数由实部和虚部两部分组成，用`j`表示。 您可以这样定义复数：

```
>>> x = 2 + 3j # where 2 is the real part and 3 is imaginary
```

4 确定类型

Python 具有`type()`内置函数，可用于确定变量的类型。

```
>>> x = 12
>>> type(x)
  <class 'int'>
```

5 Python 运算符

Python 具有不同的运算符，可让您在程序中执行所需的计算。

`+`， `-`和`*`可以正常工作，其余的运算符需要一些解释。

名称	含义	示例	结果
+	加法	15+20	35
-	减法	24.5-3.5	21.0
*	乘法	15*4	60
/	浮点除法	4/5	0.8
//	整数除法	4//5	0
**	求幂	4**2	16
%	余数	27%4	3

浮点除法（`/`）：`/`运算符进行除法并以浮点数形式返回结果，这意味着它将始终返回小数部分。 例如

```
>>> 3/2
1.5
```

整数除法（`//`）： `//`执行整数除法，即它将截断答案的小数部分并仅返回整数。

```
>>> 3//2
1
```

幂运算符（`**`）： 此运算符有助于计算`ˆb`（a 的 b 次幂）。 让我们举个例子：

```
>>> 2 ** 3 # is same as 2 * 2 * 2
8
```

余数运算符（`%`）： `%`运算符也称为余数或模数运算符。 该运算符除法后返回余数。 例如：

```
>>> 7 % 2
1
```

6 运算符优先级

在 python 中，每个表达式都使用运算符优先级进行求值。 让我们以一个例子来阐明它。

```
>>> 3 * 4 + 1
```

在上面的表达式中，将对哪个运算进行第一个加法或乘法运算？ 为了回答这样的问题，我们需要在 python 中引用运算符优先级列表。 下图列出了 python 优先级从高到低的顺序。

Operator	Description
`()`	Parentheses (grouping)
`f(args...)`	Function call
`x[index:index]`	Slicing
`x[index]`	Subscription
`x.attribute`	Attribute reference
`**`	Exponentiation
`~x`	Bitwise not
`+x, -x`	Positive, negative
`*, /, %`	Multiplication, division, remainder
`+, -`	Addition, subtraction
`<<, >>`	Bitwise shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	Comparisons, membership, identity
`not x`	Boolean NOT
`and`	Boolean AND
`or`	Boolean OR
`lambda`	Lambda expression

如您在上表中所见`*`在`+`之上，因此`*`将首先出现，然后加法。 因此，以上表达式的结果将为`13`。

```
>>> 3 * 4 + 1
>>> 13

```

让我们再举一个例子来说明另一个概念。

```
>>> 3 + 4 - 2
```

在以上表达式中，将首先进行加法或减法。 从表 `+` 和 `-` 的优先级相同，然后将它们从左到右进行求值，即先加法，然后减法。

```
>>> 3 + 4 - 2
>>> 5
```

该规则的唯一例外是赋值运算符（ `=` ），它从右到左出现。

```
a = b = c
```

您可以使用括号 `()` 更改优先级，例如

```
>> 3 * (4 + 1)
>> 15
```

从优先级表中可以看出， `()` 具有最高优先级，因此在表达式 `3 * (4 + 1)` 中，先求值 `(4 + 1)` ，然后相乘。 因此，您可以使用 `()` 更改优先级。

7 复合赋值

这些运算符使您可以编写快捷方式分配语句。 例如：

```
>>> count = 1
>>> count = count + 1
>>> count
2

```

通过使用增强分配运算符，我们可以将其编写为：

```
>>> count = 1
>>> count += 1
>>> count
2

```

类似地，您可以将 `-` ， `%` ， `//` ， `/` ， `*` 和 `**` 与赋值运算符一起使用以构成扩展赋值运算符。

运算符	名称	示例	等价于
`+=`	加法赋值	`x += 4`	`x = x + 4`
`-=`	减法赋值	`x -= 2`	`x = x - 2`
`*=`	乘法赋值	`x *= 5`	`x = x * 5`
`/=`	除法赋值	`x /= 5`	`x = x / 5`
`//=`	整数除法赋值	`x //= 5`	`x = x // 5`
`%=`	余数赋值	`x %= 5`	`x = x % 5`
`**=`	指数赋值	`x **= 5`	`x = x ** 5`

第三章 字符串

python 中的字符串是由单引号或双引号分隔的连续字符系列。 Python 没有任何单独的字符数据类型，因此它们表示为单个字符串。

1 创建字符串

```
>>> name = "tom" # a string
>>> mychar = 'a' # a character

```

您还可以使用以下语法创建字符串。

```
>>> name1 = str() # this will create empty string object
>>> name2 = str("newstring") # string object containing 'newstring'

```

```
name = "tom" # a string
mychar = 'a' # a character

print(name)
print(mychar)

name1 = str() # this will create empty string object
name2 = str("newstring") # string object containing 'newstring'

print(name1)
print(name2)
```

2 Python 中的字符串是不可变的

这对您而言意味着，一旦创建了字符串，便无法对其进行修改。 让我们以一个例子来说明这一点。

```
>>> str1 = "welcome"
>>> str2 = "welcome"

```

这里 `str1` 和 `str2` 指的是存储在内存中某个位置的相同字符串对象“`welcome`”。 您可以使用 `id()` 函数测试 `str1` 是否与 `str2` 引用相同的对象。

什么是身份证?

python 中的每个对象都存储在内存中的某个位置。 我们可以使用 `id()` 获得该内存地址。

```
>>> id(str1)
78965411
>>> id(str2)
78965411

```

由于 `str1` 和 `str2` 都指向相同的存储位置，因此它们都指向同一对象。

让我们尝试通过向其添加新字符串来修改 `str1` 对象。

```
>>> str1 += " mike"
>>> str1
welcome mike
>>> id(str1)
>>>  78965579

```

如您现在所见， `str1` 指向完全不同的内存位置，这证明了并置不会修改原始字符串对象而是创建一个新的字符串对象这一点。 同样，数字（即 `int` 类型）也是不可变的。

试试看：

```
str1 = "welcome"
str2 = "welcome"

print(id(str1),   id(str2))

str1 += " mike"

print(str1)

print(id(str1))
```

3 字符串操作

字符串索引从 `0` 开始，因此要访问字符串类型中的第一个字符：

```
>>> name[0] #
t

```

试一试：

```
name = "tom"

print(name[0])
print(name[1])
```

`+` 运算符用于连接字符串，而 `*` 运算符是字符串的重复运算符。

```
>>> s = "tom and " + "jerry"
>>> print(s)
tom and jerry

```

```
>>> s = "spamming is bad " * 3
>>> print(s)
'spamming is bad spamming is bad spamming is bad '

```

试一试：

```
s = "tom and " + "jerry"
print(s)

s = "spamming is bad " * 3
print(s)
```

4 字符串切片

您可以使用 [] 运算符（也称为切片运算符）从原始字符串中提取字符串的子集。

语法：`s[start:end]`

这将从索引`start`到索引`end - 1`返回字符串的一部分。

让我们举一些例子。

```
>>> s = "Welcome"
>>> s[1:3]
el

```

一些更多的例子。

```
>>> s = "Welcome"
>>>
>>> s[:6]
'Welcom'
>>>
>>> s[4:]
'ome'
>>>
>>> s[1:-1]
'elcom'

```

试一试：

```
s = "Welcome"

print(s[1:3])
print(s[:6])
print(s[4:])
print(s[1:-1])
```

注意：

`start` 索引和 `end` 索引是可选的。 如果省略，则 `start` 索引的默认值为 `0` ，而 `end` 的默认值为字符串的最后一个索引。

5 `ord()` 和 `chr()` 函数

`ord()` -函数返回字符的 ASCII 码。

`chr()` -函数返回由 ASCII 数字表示的字符。

```
>>> ch = 'b'
>>> ord(ch)
98
>>> chr(97)
'a'
>>> ord('A')
65

```

试一试：

```
ch = 'b'

print(ord(ch))

print(chr(97))

print(ord('A'))
```

6 Python 中的字符串函数

函数名称	函数说明
`len ()`	返回字符串的长度
`max()`	返回具有最高 ASCII 值的字符
`min()`	返回具有最低 ASCII 值的字符

```
>>> len("hello")
5
>>> max("abc")
'c'
>>> min("abc")
'a'

```

试一试：

```
print(len("hello"))

print(max("abc"))

print(min("abc"))
```

7 in 和 not in 运算符

您可以使用 in 和 not in 运算符检查另一个字符串中是否存在一个字符串。 他们也被称为会员运营商。

```
>>> s1 = "Welcome"
>>> "come" in s1
True
>>> "come" not in s1
False
>>>

```

试一试：

```
s1 = "Welcome"

print("come" in s1)

print("come" not in s1)
```

8 字符串比较

您可以使用[> ， < ， <= ， <= ， == ， != ）比较两个字符串。 Python 按字典顺序比较字符串，即使用字符的 ASCII 值。

假设您将 str1 设置为 "Mary" 并将 str2 设置为 "Mac" 。 比较 str1 和 str2 的前两个字符（ M 和 M ）。 由于它们相等，因此比较后两个字符。 因为它们也相等，所以比较了前两个字符（ r 和 c ）。 并且因为 r 具有比 c 更大的 ASCII 值，所以 str1 大于 str2 。

这里还有更多示例：

```
>>> "tim" == "tie"
False
>>> "free" != "freedom"
True
>>> "arrow" > "aron"
True
>>> "right" >= "left"
True
>>> "teeth" < "tee"
False
>>> "yellow" <= "fellow"
False
>>> "abc" > ""
True
>>>

```

试一试：

```
print("tim" == "tie")

print("free" != "freedom")

print("arrow" > "aron")

print("right" >= "left")

print("teeth" < "tee")

print("yellow" <= "fellow")

print("abc" > "")
```

9 使用 for 循环迭代字符串

字符串是一种序列类型，也可以使用 for 循环进行迭代（要了解有关 for 循环的更多信息，**请单击此处**）。

```
>>> s = "hello"
>>> for i in s:
...         print(i, end="")
hello

```

注意：

默认情况下，print() 函数用换行符打印字符串，我们通过传递名为 end 的命名关键字参数来更改此行为，如下所示。

```
print("my string", end="\n")   # this is default behavior
print("my string", end="")      # print string without a newline
print("my string", end="foo") # now print() will print foo after every string

```

试一试：

```
s = "hello"
for i in s:
      print(i, end="")
```

10 测试字符串

python 中的字符串类具有各种内置方法，可用于检查不同类型的字符串。

方法名称	方法说明
`isalnum()`	如果字符串是字母数字，则返回 `True`
`isalpha()`	如果字符串仅包含字母，则返回 `True`
`isdigit()`	如果字符串仅包含数字，则返回 `True`
`isidentifier()`	返回 `True` 是字符串是有效的标识符
`islower()`	如果字符串为小写，则返回 `True`
`isupper()`	如果字符串为大写则返回 `True`
`isspace()`	如果字符串仅包含空格，则返回 `True`

```
>>> s = "welcome to python"
>>> s.isalnum()
False
>>> "Welcome".isalpha()
True
>>> "2012".isdigit()
True
>>> "first Number".isidentifier()
False
>>> s.islower()
True
>>> "WELCOME".isupper()
True
>>> " \t".isspace()
True

```

试一试：

```
s = "welcome to python"

print(s.isalnum())
print("Welcome".isalpha())
print("2012".isdigit())
print("first Number".isidentifier())
print(s.islower())
print("WELCOME".isupper())
print("  \t".isspace())
```

11 搜索子串

方法名称	方法说明
`endwith(s1: str): bool`	如果字符串以子字符串 `s1` 结尾，则返回 `True`
`startswith(s1: str): bool`	如果字符串以子字符串 `s1` 开头，则返回 `True`
`count(s: str): int`	返回字符串中子字符串出现的次数
`find(s1): int`	返回字符串中 `s1` 起始处的最低索引，如果找不到字符串则返回 `-1`
`rfind(s1): int`	从字符串中 `s1` 的起始位置返回最高索引，如果找不到字符串则返回 `-1`

```
>>> s = "welcome to python"
>>> s.endswith("thon")
True
>>> s.startswith("good")
False
>>> s.find("come")
3
>>> s.find("become")
-1
>>> s.rfind("o")
15
>>> s.count("o")
3
>>>

```

试一试：

```
s = "welcome to python"

print(s.endswith("thon"))

print(s.startswith("good"))

print(s.find("come"))

print(s.find("become"))

print(s.rfind("o"))

print(s.count("o"))
```

12 转换字符串

方法名称	方法说明
`capitalize(): str`	返回此字符串的副本，仅第一个字符大写。
`lower(): str`	通过将每个字符转换为小写来返回字符串
`upper(): str`	通过将每个字符转换为大写来返回字符串
`title(): str`	此函数通过大写字符串中每个单词的首字母来返回字符串
`swapcase(): str`	返回一个字符串，其中小写字母转换为大写，大写字母转换为小写
`replace(old, new): str`	此函数通过用新字符串替换旧字符串的出现来返回新字符串

```
s = "string in python"
>>>
>>> s1 = s.capitalize()
>>> s1
'String in python'
>>>
>>> s2 = s.title()
>>> s2
'String In Python'
>>>
>>> s = "This Is Test"
>>> s3 = s.lower()
>>> s3
'this is test'
>>>
>>> s4 = s.upper()
>>> s4
'THIS IS TEST'
>>>
>>> s5 = s.swapcase()
>>> s5
'tHIS iS tEST'
>>>
>>> s6 = s.replace("Is", "Was")
>>> s6
'This Was Test'
>>>
>>> s
'This Is Test'
>>>

```

试一试：

```
s = "string in python"

s1 = s.capitalize()
print(s1)

s2 = s.title()
print(s2)

s = "This Is Test"
s3 = s.lower()
```

```
11
12  print(s3)
13
14  s4 = s.upper()
15  print(s4)
16
17  s5 = s.swapcase()
18  print(s5)
19
20  s6 = s.replace("Is", "Was")
21  print(s6)
22
23  print(s)
```

第四章 列表

列表类型是 python 的列表类定义的另一种序列类型。 列表允许您以非常简单的方式添加，删除或处理元素。 列表与数组非常相似。

1 在 python 中创建列表

您可以使用以下语法创建列表。

```
1  >>> l = [1, 2, 3, 4]
2
```

在此，列表中的每个元素都用逗号分隔，并用一对方括号（`[]`）包围。 列表中的元素可以是相同类型或不同类型。 例如：

```
1  l2 = ["this is a string", 12]
2
```

创建列表的其他方式。

```
1  list1 = list() # Create an empty list
2  list2 = list([22, 31, 61]) # Create a list with elements 22, 31, 61
3  list3 = list(["tom", "jerry", "spyke"]) # Create a list with strings
4  list5 = list("python") # Create a list with characters p, y, t, h, o, n
5
```

注意：

列表是可变的。

2 访问列表中的元素

您可以使用索引运算符（`[]`）访问列表中的各个元素。 列表索引从`0`开始。

```
>>> l = [1,2,3,4,5]
>>> l[1] # access second element in the list
2
>>> l[0] # access first element in the list
1

```

3 常用列表操作

方法名称	描述
`x in s`	如果元素 `x` 在序列 `s` 中，则为 `True`，否则为 `False`
`x not in s`	如果元素 `x` 不在序列 `s` 中，则为 `True`，否则为 `False`
`s1 + s2`	连接两个序列 `s1` 和 `s2`
`s * n`， `n * s`	连接序列 `s` 的 `n` 个副本
`s[i]`	序列 `s` 的第 `i` 个元素。
`len(s)`	序列 `s` 的长度，也就是元素数量。
`min(s)`	序列 `s` 中最小的元素。
`max(s)`	序列 `s` 中最大的元素。
`sum(s)`	序列 `s` 中所有数字的总和。
`for` 循环	在 `for` 循环中从左到右遍历元素。

4 列表函数示例

```
>>> list1 = [2, 3, 4, 1, 32]
>>> 2 in list1
True
>>> 33 not in list1
True
>>> len(list1) # find the number of elements in the list
5
>>> max(list1) # find the largest element in the list
32
>>> min(list1) # find the smallest element in the list
1
>>> sum(list1) # sum of elements in the list
42

```

5 列表切片

切片运算符（`[start:end]`）允许从列表中获取子列表。 它的工作原理类似于字符串。

```
>>> list = [11,33,44,66,788,1]
>>> list[0:5] # this will return list starting from index 0 to index 4
[11,33,44,66,788]

```

```
>>> list[:3]
[11,33,44]

```

类似于字符串 `start` 的索引是可选的，如果省略，它将为 `0` 。

```
>>> list[2:]
[44,66,788,1]

```

`end` 索引也是可选的，如果省略，它将被设置为列表的最后一个索引。

注意:

如果为 `start >= end` ，则 `list[start : end]` 将返回一个空列表。 如果 `end` 指定的位置超出列表的 `end` ，则 Python 将使用 `end` 的列表长度。

6 列表中的 `+` 和 `*` 运算符

`+` 运算符加入两个列表。

```
>>> list1 = [11, 33]
>>> list2 = [1, 9]
>>> list3 = list1 + list2
>>> list3
[11, 33, 1, 9]

```

`*` 操作符复制列表中的元素。

```
>>> list4 = [1, 2, 3, 4]
>>> list5 = list4 * 3
>>> list5
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]

```

7 `in` 或 `not in` 运算符

`in` 运算符用于确定列表中是否存在元素。 成功则返回 `True` ； 失败则返回 `False` 。

```
>>> list1 = [11, 22, 44, 16, 77, 98]
>>> 22 in list1
True

```

同样， `not in` 与 `in` 运算符相反。

```
>>> 22 not in list1
False

```

8 使用 `for` 循环遍历列表

如前所述，列表是一个序列并且也是可迭代的。 意味着您可以使用 `for` 循环遍历列表的所有元素。

```
>>> list = [1,2,3,4,5]
>>> for i in list:
... print(i, end=" ")
1 2 3 4 5

```

9 常用列表方法和返回类型

方法	描述
`append(x: object): None`	在列表的末尾添加元素 `x` 并返回 `None` 。
`count(x: object): int`	返回元素 `x` 在列表中出现的次数。
`append(l: list): None`	将 `l` 中的所有元素附加到列表中并返回 `None` 。
`index(x: object): int`	返回列表中第一次出现的元素 `x` 的索引
`insert(index: int, x: object): None`	在给定索引处插入元素 `x` 。 请注意，列表中的第一个元素具有索引 `0` 并返回 `None` 。
`remove(x: object): None`	从列表中删除第一次出现的元素 `x` 并返回 `None`
`reverse(): None`	反转列表并返回 `None`
`sort(): None`	按升序对列表中的元素进行排序并返回 `None` 。
`pop(i): object`	删除给定位置的元素并返回它。 参数 `i` 是可选的。 如果未指定，则 `pop()` 删除并返回列表中的最后一个元素。
`copy():object`	返回一个复制
`clear()`	移除所有元素 `del a[:]`
`list.extend(iterable)`	使用可迭代对象中的所有元素来扩展列表。

```
>>> list1 = [2, 3, 4, 1, 32, 4]
>>> list1.append(19)
>>> list1
[2, 3, 4, 1, 32, 4, 19]
>>> list1.count(4) # Return the count for number 4
2
>>> list2 = [99, 54]
```

```
>>> list1.extend(list2)
>>> list1
[2, 3, 4, 1, 32, 4, 19, 99, 54]
>>> list1.index(4) # Return the index of number 4
2
>>> list1.insert(1, 25) # Insert 25 at position index 1
>>> list1
[2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
>>>
>>> list1 = [2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
>>> list1.pop(2)
3
>>> list1
[2, 25, 4, 1, 32, 4, 19, 99, 54]
>>> list1.pop()
54
>>> list1
[2, 25, 4, 1, 32, 4, 19, 99]
>>> list1.remove(32) # Remove number 32
>>> list1
[2, 25, 4, 1, 4, 19, 99]
>>> list1.reverse() # Reverse the list
>>> list1
[99, 19, 4, 1, 4, 25, 2]
>>> list1.sort() # Sort the list
>>> list1
[1, 2, 4, 4, 19, 25, 99]
>>>

```

10 列表推导式

注意：

本主题需要具有 **Python 循环** 的使用知识。

列表理解为创建列表提供了一种简洁的方法。 它由包含表达式的方括号组成，后跟 `for` 子句，然后是零个或多个 `for` 或 `if` 子句。

这里有些例子：

```
>>> list1 = [ x for x in range(10) ]
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>>
>>> list2 =  [ x + 1 for x in range(10) ]
>>> list2
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>>
>>>
>>>  list3 = [ x for x in range(10) if x % 2 == 0 ]
>>> list3
[0, 2, 4, 6, 8]
>>>
>>>
>>>  list4 = [ x *2 for x in range(10) if x % 2 == 0 ]
```

```
[0, 4, 8, 12, 16]

```

11 列表作为栈使用

列表方法使得列表作为堆栈非常容易，最后一个插入，最先取出（“后进先出”）。要添加一个元素到堆栈的顶端，使用 append() 。要从堆栈顶部取出一个元素，使用 pop() ，不用指定索引。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]
```

12 列表作为队列使用

列表也可以用作队列，其中先添加的元素被最先取出 (“先进先出”)；然而列表用作这个目的相当低效。因为在列表的末尾添加和弹出元素非常快，但是在列表的开头插入或弹出元素却很慢 (因为所有的其他元素都必须移动一位)。

若要实现一个队列，可使用 collections.deque，它被设计成可以快速地从两端添加或弹出元素。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'])
```

第五章 字典

字典是一种 python 数据类型，用于存储键值对。 它使您可以使用键快速检索，添加，删除，修改值。字典与我们在其他语言上称为关联数组或哈希的非常相似。

注意:

字典是可变的。

1 创建字典

可以使用一对大括号（ {} ）创建字典。 字典中的每个项目都由一个键，一个冒号，一个值组成。 每个项目都用逗号（ , ）分隔。 让我们举个例子。

```
friends = {
'tom' : '111-222-333',
'jerry' : '666-33-111'
}

```

这里 friends 是有两个项目的字典。 需要注意的一点是，键必须是可哈希的类型，但是值可以是任何类型。 字典中的每个键都必须是唯一的。

```
>>> dict_emp = {} # this will create an empty dictionary

```

- dict() 构造函数可以直接从键值对序列里创建字典。

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

- 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}
```

- 当关键字是简单字符串时，有时直接通过关键字参数来指定键值对更方便

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

2 检索，修改和向字典中添加元素

要从字典中获取项目，请使用以下语法：

```
>>> dictionary_name['key']

```

```
>>> friends['tom']
'111-222-333'

```

如果字典中存在键，则将返回值，否则将引发 KeyError 异常。 要添加或修改项目，请使用以下语法：

```
>>> dictionary_name['newkey'] = 'newvalue'

```

```
>>> friends['bob'] = '888-999-666'
>>> friends
  {'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}

```

3 从字典中删除项目

```
>>> del dictionary_name['key']

```

```
>>>    del friends['bob']
>>>   friends
{'tom': '111-222-333', 'jerry': '666-33-111'}

```

如果找到键，则该项目将被删除，否则将抛出 `KeyError` 异常。

4 遍历字典中的项目

您可以使用 `for` 循环遍历字典中的元素。

```
>>> friends = {
... 'tom' : '111-222-333',
... 'jerry' : '666-33-111'
...}
>>>
>>> for key in friends:
...       print(key, ":", friends[key])
...
tom : 111-222-333
jerry : 666-33-111
>>>
>>>

```

5 查找字典的长度

您可以使用 `len()` 函数查找字典的长度。

```
>>> len(friends)
2

```

6 `in` 和 `not in` 运算符

`in` 和 `not in` 运算符检查字典中是否存在键。

```
>>> 'tom' in friends
True
>>> 'tom' not in friends
False

```

7 字典中的相等测试

`==` 和 `!=` 运算符告诉字典是否包含相同的项目。

```
>>> d1 = {"mike":41, "bob":3}
>>> d2 = {"bob":3, "mike":41}
>>> d1 == d2
True
>>> d1 != d2
False
>>>

```

注意:

您不能使用 `<`, `>`, `>=`, `<=` 等其他关系运算符来比较字典。

8 字典方法

Python 提供了几种内置的方法来处理字典。

方法	描述
`popitem()`	返回字典中随机选择的项目，并删除所选项目。
`clear()`	删除字典中的所有内容
`keys()`	以元组形式返回字典中的键
`values()`	以元组形式返回字典中的值
`get(key)`	键的返回值，如果找不到键，则返回 `None`，而不是引发 `KeyError` 异常
`pop(key)`	从字典中删除该项目，如果找不到该键，则会抛出 `KeyError`

```
>>> friends = {'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}
>>>
>>> friends.popitem()
('tom', '111-222-333')
>>>
>>>  friends.clear()
>>>
>>>  friends
{}
>>>
>>>  friends = {'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}
>>>
>>> friends.keys()
dict_keys(['tom', 'bob', 'jerry'])
>>>
>>> friends.values()
dict_values(['111-222-333', '888-999-666', '666-33-111'])
>>>
>>>  friends.get('tom')
'111-222-333'
```

```
>>>
>>>  friends.get('mike', 'Not Exists')
'Not Exists'
>>>
>>>  friends.pop('bob')
'888-999-666'
>>>
>>> friends
{'tom': '111-222-333', 'jerry': '666-33-111'}

```

第六章 元组

在 Python 中，元组与列表非常相似，但是一旦创建了元组，就无法添加，删除，替换和重新排序元素。

注意：

元组是不可变的。

1 创建一个元组

```
>>> t1 = () # creates an empty tuple with no data
>>>
>>> t2 = (11,22,33)
>>>
>>> t3 = tuple([1,2,3,4,4]) # tuple from array
>>>
>>> t4 = tuple("abc") # tuple from string

```

2 元组函数

元组也可以使用 `max()` ， `min()` ， `len()` 和 `sum()` 之类的函数。

```
>>> t1 = (1, 12, 55, 12, 81)
>>> min(t1)
1
>>> max(t1)
81
>>> sum(t1)
161
>>> len(t1)
5

```

3 元组迭代

元组可使用 `for` 循环进行迭代， **在此处了解有关 for 循环的更多信息** 。

```
>>> t = (11,22,33,44,55)
>>> for i in t:
...         print(i, end=" ")
>>> 11 22 33 44 55

```

4 元组切片

切片运算符在元组中的作用与在列表和字符串中的作用相同。

```
>>> t = (11,22,33,44,55)
>>> t[0:2]
(11,22)

```

5 `in` 和 `not in` 运算符

您可以使用 `in` 和 `not in` 运算符检查元组中项的存在，如下所示。

```
>>> t = (11,22,33,44,55)
>>> 22 in t
True
>>> 22 not in t
False

```

第七章 集合

Python也包含有 集合 类型。集合是由不重复元素组成的无序的集。它的基本用法包括成员检测和消除重复元素。集合对象也支持像 联合，交集，差集，对称差分等数学运算。

花括号或 set() 函数可以用来创建集合。注意：要创建一个空集合你只能用 set() 而不能用 {}，因为后者是创建一个空字典，这种数据结构我们会在下一节进行讨论。

以下是一些简单的示例

```
>>>
>>> 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 a or b or both
{'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'}

```

类似于 列表推导式，集合也支持推导式形式

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

第八章 数据类型转换

偶尔，您会希望将一种类型的数据类型转换为另一种类型。 数据类型转换也称为类型转换。

1 将 int 转换为 float

要将 `int` 转换为 `float`，可以使用 `float()` 函数。

```
>>> i = 10
>>> float(i)
10.0

```

2 将 float 转换为 int

要将 `float` 转换为 `int`，您需要使用 `int()` 函数。

```
>>> f = 14.66
>>> int(f)
14

```

3 将字符串转换为 int

要将 string 转换为 int，请使用 int() 函数。

```
>>> s = "123"
>>> int(s)
123

```

提示：

如果字符串包含非数字字符，则 int() 将引发 ValueError 异常。

4 将数字转换为字符串

要将数字转换为字符串，请使用 str() 函数。

```
>>> i = 100
>>> str(i)
"100"
>>> f = 1.3
str(f)
'1.3'

```

5 舍入数字

四舍五入数字是通过 round() 函数完成的。

语法： round(number[, ndigits])

```
>>> i = 23.97312
>>> round(i, 2)
23.97
```

第九章 控制流

1 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
```

2 for语句

对任意序列进行迭代（例如列表或字符串），条目的迭代顺序与它们在序列中出现的顺序一致。

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

在遍历同一个集合时修改该集合的代码可能很难获得正确的结果。通常，更直接的做法是循环遍历该集合的副本或创建新集合：

```
# Strategy:  Iterate over a copy
for user, status in users.copy().items():
      if status == 'inactive':
            del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
      if status == 'active':
            active_users[user] = status
```

3 else循环语句

try 语句中的 else 子句会在未发生异常时执行，而循环中的 else 子句则会在未发生 break 时执行。循环语句可能带有 else 子句；它会在循环耗尽了可迭代对象 (使用 for) 或循环条件变为假值 (使用 while) 时被执行，但不会在循环被 break 语句终止时被执行。

```
>>> 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
```

4 break语句

break 语句，和 C 中的类似，用于跳出最近的 for 或 while 循环.

5 continue语句

continue 语句也是借鉴自 C 语言，表示继续循环中的下一次迭代:

```
>>> for num in range(2, 10):
...         if num % 2 == 0:
...                 print("Found an even number", num)
...                 continue
...         print("Found an odd number", num)
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

6 pass语句

pass 语句什么也不做。当语法上需要一个语句，但程序需要什么动作也不做时，可以使用它。例如:

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

第十章 函数

1 定义函数

```
>>> 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
```

> 参数都是引用传递。

- 关键字 def 引入一个函数 定义。它必须后跟函数名称和带括号的形式参数列表。构成函数体的语句从下一行开始，并且必须缩进。
- 函数体的第一个语句可以（可选的）是字符串文字。用来生成文档
- 函数的 执行 会引入一个用于函数局部变量的新符号表。 更确切地说，函数中所有的变量赋值都将存储在局部符号表中；而变量引用会首先在局部符号表中查找，然后是外层函数的局部符号表，再然后是全局符号表，最后是内置名称的符号表。
- 在函数被调用时，实际参数（实参）会被引入被调用函数的本地符号表中；因此，实参是通过 按值调用 传递的（其中 值 始终是对象 引用 而不是对象的值）。1 当一个函数调用另外一个函数时，将会为该调用创建一个新的本地符号表。

2 默认参数(可选参数)（函数定义）

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    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 ValueError('invalid user response')
        print(reminder)
```

这个函数可以通过几种方式调用:

- 只给出必需的参数：ask_ok('Do you really want to quit?')
- 给出一个可选的参数：ask_ok('OK to overwrite the file?', 2)
- 或者给出所有的参数：ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

> 多次调用默认参数会共享

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

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

3 位置参数和关键字参数（函数调用）

也可以使用形如 kwarg=value 的 关键字参数 来调用函数。例如下面的函数:

```
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, "!")
```

接受一个必需的参数（voltage）和三个可选的参数（state, action，和 type）。这个函数可以通过下面的任何一种方式调用:

```
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
```

但下面的函数调用都是无效的:

```
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其中一个参数匹配（比如 actor 不是函数 parrot 的有效参数），它们的顺序并不重要。
- 不能对同一个参数多次赋值。

4 复合参数（函数定义）

当存在一个形式为 `**name` 的最后一个形参时，它会接收一个字典，其中包含除了与已有形参相对应的关键字参数以外的所有关键字参数。

这可以与一个形式为 `*name`，接收一个包含除了已有形参列表以外的位置参数的 元组 的形参组合使用 (*name 必须出现在 **name 之前。)

例如，如果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)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

它可以像这样调用:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

当然它会打印:

```
-- 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.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

注意打印时关键字参数的顺序保证与调用函数时提供它们的顺序是相匹配的。

5 特殊参数（函数定义）

默认情况下，函数的参数传递形式可以是位置参数或是显式的关键字参数。 为了确保可读性和运行效率，限制允许的参数传递形式是有意义的，这样开发者只需查看函数定义即可确定参数项是仅按位置、按位置也按关键字，还是仅按关键字传递。

函数的定义看起来可以像是这样:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      ----------     ----------       ----------
        |             |                  |
        |        Positional or keyword    |
        |                                  - Keyword only
         -- Positional only
```

- / 和 * 是可选的。 如果使用这些符号则表明可以通过何种形参将参数值传递给函数：仅限位置、位置或关键字，以及仅限关键字。 关键字形参也被称为命名形参。
- 如果函数定义中未使用 / 和 *，则参数可以按位置或按关键字传递给函数。

5.1 函数实例

- 请考虑以下示例函数定义并特别注意 / 和 * 标记:

```
>>>
>>> def standard_arg(arg):
...       print(arg)
...
>>> def pos_only_arg(arg, /):
...       print(arg)
...
>>> def kwd_only_arg(*, arg):
...       print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...       print(pos_only, standard, kwd_only)
```

- 第一个函数定义 standard_arg 是最常见的形式，对调用方式没有任何限制，参数可以按位置也可以按关键字传入:

```
>>>
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

- 第二个函数 pos_only_arg 在函数定义中带有 /，限制仅使用位置形参。：

```
>>>
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got an unexpected keyword argument 'arg'
```

- 第三个函数 kwd_only_args 在函数定义中通过 * 指明仅允许关键字参数:

```
>>>
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

- 而最后一个则在同一函数定义中使用了全部三种调用方式:

```
>>>
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got an unexpected keyword argument 'pos_only'
```

- 最后，请考虑这个函数定义，它的位置参数 name 和 **kwds 之间由于存在关键字名称 name 而可能产生潜在冲突:

```
def foo(name, **kwds):
      return 'name' in kwds

```

任何调用都不可能让它返回 True，因为关键字 'name' 将总是绑定到第一个形参。 例如:

```
>>>
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

- 但使用 / (仅限位置参数) 就可能做到，因为它允许 name 作为位置参数，也允许 'name' 作为关键字参数的关键字名称:

```
def foo(name, /, **kwds):
      return 'name' in kwds
>>> foo(1, **{'name': 2})
True
```

- 换句话说，仅限位置形参的名称可以在 **kwds 中使用而不产生歧义。

5.2 使用指导

- 如果你希望形参名称对用户来说不可用，则使用仅限位置形参。 这适用于形参名称没有实际意义，以及当你希望强制规定调用时的参数顺序，或是需要同时收受一些位置形参和任意关键字形参等情况。
- 当形参名称有实际意义，以及显式指定形参名称可使函数定义更易理解，或者当你想要防止用户过于依赖传入参数的位置时，则使用仅限关键字形参。
- 对于 API 来说，使用仅限位置形参可以防止形参名称在未来被修改时造成破坏性的 API 变动。

6 解包参数列表（函数调用）

当参数已经在列表或元组中但要为需要单独位置参数的函数调用解包时，会发生相反的情况。例如，内置的 range() 函数需要单独的 start 和 stop 参数。如果它们不能单独使用，可以使用 * 操作符 来编写函数调用以便从列表或元组中解包参数:

```
>>>
>>> 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]
```

同样的方式，字典可使用 ** 操作符 来提供关键字参数:

```
>>>
>>> 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 !
```

> 使用*和**来对元组参数和字典参数进行解包，与定义复合参数的函数相对应。

7 Lambda表达式（函数定义）

可以用 lambda 关键字来创建一个小的匿名函数。这个函数返回两个参数的和： lambda a, b: a+b 。Lambda函数可以在需要函数对象的任何地方使用。它们在语法上限于单个表达式。从语义上来说，它们只是正常函数定义的语法糖。与嵌套函数定义一样，lambda函数可以引用所包含域的变量:

```
>>>
>>> def make_incrementor(n):
...         return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

第十一章 类

特点:

- 类提供了一种组合数据和功能的方法。 创建一个新类意味着创建一个新的对象 类型，从而允许创建一个该类型的新 实例 。 每个类的实例可以拥有保存自己状态的属性。 一个类的实例也可以有改变自己状态的（定义在类中的）方法。
- 类继承机制允许多个基类，派生类可以覆盖它基类的任何方法，一个方法可以调用基类中相同名称的的方法。对象可以包含任意数量和类型的数据。和模块一样，类也拥有 Python 天然的动态特性：它们在运行时创建，可以在创建后修改。

> 只有自定义的类的对象才会有同态属性，系统内建的对象，不允许有动态属性。

1 名称和对象

对象具有个性，多个名称（在多个作用域内）可以绑定到同一个对象。这在其他语言中称为别名。

> 变量存储对象的引用。对象变量作为参数传递时，传递的是对象的引用。

2 类定义语法

最简单的类定义看起来像这样:

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

3 类的对象

类对象支持两种操作：属性引用和实例化。

属性引用

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

```
class MyClass:
      """A simple example class"""
      i = 12345

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

那么 MyClass.i 和 MyClass.f 就是有效的属性引用，将分别返回一个整数和一个函数对象。 类属性也可以被赋值，因此可以通过赋值来更改 MyClass.i 的值。

实例化

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

x = MyClass()
创建类的新 实例 并将此对象分配给局部变量 x。

实例化操作（“调用”类对象）会创建一个空对象。 许多类喜欢创建带有特定初始状态的自定义实例。 为此类定义可能包含一个名为 **init**() 的特殊方法，就像这样:

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

当一个类定义了 **init**() 方法时，类的实例化操作会自动为新创建的类实例发起调用 **init**()。 因此在这个示例中，可以通过以下语句获得一个经初始化的新实例:

```
x = MyClass()
```

当然， **init**() 方法还可以有额外参数以实现更高灵活性。 在这种情况下，提供给类实例化运算符的参数将被传递给 **init**()。 例如，：

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

4 继承

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

```
class DerivedClassName(BaseClassName):
      <statement-1>
     .
     .
     .
      <statement-N>
```

名称 BaseClassName 必须定义于包含派生类定义的作用域中。 也允许用其他任意表达式代替基类名称所在的位置。 这有时也可能会用得上，例如，当基类定义在另一个模块中的时候:

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

5 多重继承

Python也支持一种多重继承。 带有多个基类的类定义语句如下所示:

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

6 私有变量

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

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

名称改写有助于让子类重载方法而不破坏类内方法调用。例如:

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

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

      __update = update    # private copy of original update() method

class MappingSubclass(Mapping):

      def update(self, keys, values):
            # provides new signature for update()
            # but does not break __init__()
            for item in zip(keys, values):
                  self.items_list.append(item)
```

7 迭代器

到目前为止，您可能已经注意到大多数容器对象都可以使用 for 语句:

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

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

```
>>>
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
      next(it)
StopIteration
```

看过迭代器协议的幕后机制，给你的类添加迭代器行为就很容易了。 定义一个 **iter**() 方法来返回一个带有 **next**() 方法的对象。 如果类已定义了 **next**()，则 **iter**() 可以简单地返回 self:

```
class Reverse:
      """Iterator for looping over a sequence backwards."""
      def __init__(self, data):
            self.data = data
            self.index = len(data)

      def __iter__(self):
            return self

      def __next__(self):
            if self.index == 0:
                  raise StopIteration
            self.index = self.index - 1
            return self.data[self.index]
>>>
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
```

```
>>> for char in rev:
...         print(char)
...
m
a
p
s
```

8 生成器

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

```
def reverse(data):
      for index in range(len(data)-1, -1, -1):
            yield data[index]
>>>
>>> for char in reverse('golf'):
...         print(char)
...
f
l
o
g
```

可以用生成器来完成的操作同样可以用前一节所描述的基于类的迭代器来完成。 但生成器的写法更为紧凑，因为它会自动创建 **iter**() 和 **next**() 方法。

9 生成器表达式

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

示例:

```
>>>
>>> sum(i*i for i in range(10))                         # sum of squares
285

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

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

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

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