

Python 3基础学习笔记

2014年8月23日

- 作者：李松
- CSDN博客：<http://blog.csdn.net/lisonglisonglisong>
- GitHub博客：<http://songlee24.github.com>

前言

Python是一门强大的解释型、面向对象的高级程序设计语言，它优雅、简单、可移植、易扩展，可用于桌面应用、系统编程、数据库编程、网络编程、web开发、图像处理、人工智能、数学应用、文本处理等等。

这个学习笔记比较基础但比较系统，适合Python的初学者用以快速入门。它讲解了Python 3.x的基本语法和高级特性，从基本数据类型、运算符、流程控制、函数、文件IO到模块、异常处理和面向对象，并且附上了很多经过测试的代码帮助读者去理解。

相信看完这个学习笔记，你会对Python有一个整体的概念，这会激起你对这门语言的兴趣，那时候你可以进一步去深入了解Python标准库，然后可以找几个Python小项目练练手。

注：

1. 本学习笔记以**python 3.4.1**版本为基础。
2. 代码测试环境为 **linux-fedora 20**。
3. 语言水平有限，表述如有不准确的地方，敬请指正。

第一篇 HelloWorld

对于新手一般会遇到一个问题：学习**Python 2**还是**Python 3**呢？对于我个人而言，我开始学习的时候是个完全的新手，没有历史包袱，所以我直接学习Python 3。我相信在未来几年，Python 3会逐步取代Python 2成为主流。

Python是一门解释型、面向对象、动态数据类型的高级程序设计语言，更多的介绍看[《官方文档》](#)。Python让程序更加的紧凑、可读性更强，用Python写的程序通常比同样的C、C++或Java程序要短得多，这是因为以下几个原因：

- 高级数据结构使你能够在单条的语句中表达复杂的操作。
- 代码块的组织依赖于缩进而不是开始/结束符，例如{ }。
- 参数或变量不需要声明。

OK！下面开始让我们进入Python的世界吧！

一、第一个HelloWorld程序

1、交互模式

交互模式即命令行模式，在Linux终端输入 `$python` 即可进入Python交互模式，主提示符`>>>`提示你输入命令：

```
$ python
Python 3.3.2 (default, Jun 30 2014, 17:20:03)
[GCC 4.8.3 20140624 (Red Hat 4.8.3-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

输入多行结构时就需要从属提示符了，例如下面这个if语句：

```
>>> flag=True
>>> if flag:
...     print("Hello World!")
...
Hello World!
>>>
```

注：在Python 3.x中，print是一个函数，所以必须使用括号输出。

2、脚本文件

我们也可以将代码写到一个.py文件中：

```
print ("Hello World!")
```

然后用python命令执行该脚本文件：

```
$ python hello.py
Hello World!
```

3、可执行脚本

在类Unix系统中，Python脚本可以像Shell脚本那样直接执行，通过在脚本文件开头添加一行：

```
#!/usr/bin/env python
```

然后通过chmod命令修改权限为可执行：

```
$ chmod +x hello.py
```

执行：

```
$ ./hello.py
Hello World!
```

二、基础语法

1、编码

默认情况下，Python 3源码文件以 UTF-8 编码，所有字符串都是 unicode 字符串。当然你也可以为源码文件指定不同的编码：

```
# -*- coding: cp-1252 -*-
```

2、标识符

在Python 3中，非-ASCII 标识符也是允许的了。但最好还是只使用英文、数字、下划线作为标识符，并且不能以数字开头。（区分大小写）

3、python保留字

保留字即关键字，我们不能把它们用作任何标识符名称。Python的标准库提供了一个keyword module，可以输出当前版本的所有关键字：

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

4、注释

Python中单行注释以#开头，多行注释用三个单引号（'''）或者三个双引号（"""）将注释括起来。

5、行与缩进

python最具特色的就是使用缩进来表示代码块。缩进的空格数是可变的，但是同一个代码块的语句必须包含相同的缩进空格数。

附：

在现在的Unix/Linux系统中都会内置Python解释器，在我的Fedora20系统中就同时装了python2和python3。在终端输入 `$python` 会使用python 2.x的解释器，输入 `$python3` 会使用python 3.x的解释器。现在我想修改一下软链接，让 `$python` 命令默认使用python 3.x解释器，`$python2` 命令使用python 2.x解释器。怎么做？

软链接：一种特殊的文件，该文件的内容是指向另一个文件的位置或路径。它不占用磁盘空间，类似于Windows操作系统中的快捷方式。

硬链接：`ln [参数] 源文件 目标文件`
软链接：`ln -s [源文件或目录] [目标文件或目录]`

`ln`命令的功能是为某一个文件在另外一个位置建立一个同步的链接（软链接或者硬链接），现在我要修改python的软链接：

```
$ sudo ln -s /usr/bin/python2.7 /usr/bin/python2 # 创建python2软链接
$ sudo rm /usr/bin/python # 删除原来的python软链接
$ sudo ln -s /usr/bin/python3.3 /usr/bin/python # 创建新的python软链接
```

修改了软链接以后的一个伴随的问题是：有些用python写的命令不能执行了，因为python2和python3的语法不兼容。比如yum命令，它是python写的，从yum文件开头的 `#!/usr/bin/python` 可以看出来。因为python软链接被修改为指向python 3.x解释器，所以执行yum命令时会报语法错误。

我们可以通过修改yum文件来修复这个问题，既然之前我们已经创建了python2软链接指向python 2.x解释器，所以我们修改yum文件开头为 `#!/usr/bin/python2` 就可以了，这样在执行yum命令时就会使用python 2.x解释器而不是python 3.x的解释器了。

第二篇 基本数据类型

Python中的变量不需要声明。每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。在Python中，变量就是变量，它没有类型，我们所说的“类型”是变量所指的内存中对象的类型。Python 3中有六个标准的数据类型：

- Numbers (数字)
- String (字符串)
- List (列表)
- Tuple (元组)
- Sets (集合)
- Dictionaries (字典)

本篇主要先介绍这几种数据类型的定义和它们之间的联系与区别。

一、Numbers

Python 3支持**int**、**float**、**bool**、**complex** (复数)。数值类型的赋值和计算都是很直观的，就像大多数语言一样。内置的 `type()` 函数可以用来查询变量所指的对象类型。

```
>>> a, b, c, d = 20, 5.5, True, 4+3j
>>> print(type(a), type(b), type(c), type(d))
<class 'int'> <class 'float'> <class 'bool'> <class 'complex'>
```

数值运算：

```
>>> 5 + 4 # 加法
9
>>> 4.3 - 2 # 减法
2.3
>>> 3 * 7 # 乘法
21
>>> 2 / 4 # 除法，得到一个浮点数
0.5
>>> 2 // 4 # 除法，得到一个整数
0
>>> 17 % 3 # 取余
2
>>> 2 ** 5 # 乘方
32
```

要点：

- 1、Python可以同时为多个变量赋值，如 `a, b = 1, 2`。
- 2、一个变量可以通过赋值指向不同类型的对象。
- 3、数值的除法 (/) 总是返回一个浮点数，要获取整数使用//操作符。
- 4、在混合计算时，Python会把整型转换成为浮点数。

二、Strings

Python中的字符串str用单引号(')或双引号(")括起来，同时使用反斜杠(\)转义特殊字符。

```
>>> s = 'Yes,he doesn\'t'
>>> print(s, type(s), len(s))
Yes,he doesn't <class 'str'> 14
```

如果你不想让反斜杠发生转义，可以在字符串前面添加一个r，表示原始字符串：

```
>>> print('C:\some\name')
C:\some
ame
>>> print(r'C:\some\name')
C:\some\name
```

另外，反斜杠可以作为续行符，表示下一行是上一行的延续。还可以使用 `"""..."""` 或者 `'''...'''` 表示跨越多行的字符串。

字符串可以使用 + 运算符串连接在一起，或者用 * 运算符重复：

```
>>> print('str'+ 'ing', 'my'*3)
string mymymy
```

Python中的字符串有两种索引方式，第一种是从左往右，从0开始依次增加；第二种是从右往左，从-1开始依次减少。注意，没有单独的字符类型，一个字符就是长度为1的字符串。

```
>>> word = 'Python'
>>> print(word[0], word[5])
P n
>>> print(word[-1], word[-6])
n P
```

还可以对字符串进行切片，获取一段子串。用冒号分隔两个索引，形式为 `变量[头下标:尾下标]`。截取的范围是前闭后开的，并且两个索引都可以省略：

```
>>> word = 'ilovepython'
>>> word[1:5]
'love'
>>> word[: ]
'ilovepython'
>>> word[5: ]
'python'
>>> word[-10:-6]
'love'
```

与C字符串不同的是，Python字符串不能被改变。向一个索引位置赋值，比如 `word[0] = 'm'` 会导致错误。

要点：

- 1、反斜杠可以用来转义，使用r可以让反斜杠不发生转义。
- 2、字符串可以用+运算符连接在一起，用*运算符重复。
- 3、Python中的字符串有两种索引方式，从左往右以0开始，从右往左以-1开始。
- 4、Python中的字符串不能改变。

三、List

List（列表）是 Python 中使用最频繁的数据类型。列表是写在方括号之间、用逗号分隔开的元素列表。列表中元素的类型可以不相同：

```
>>> a = ['him', 25, 100, 'her']
>>> print(a)
['him', 25, 100, 'her']
```

和字符串一样，列表同样可以被索引和切片，列表被切片后返回一个包含所需元素的新列表。详细的在这里就不赘述了。

列表还支持串联操作，使用+操作符：

```
>>> a = [1, 2, 3, 4, 5]
>>> a + [6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

与Python字符串不一样的是，列表中的元素是可以改变的：

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> a[0] = 9
>>> a[2:5] = [13, 14, 15]
>>> a
[9, 2, 13, 14, 15, 6]
>>> a[2:5] = [] # 删除
>>> a
[9, 2, 6]
```

List内置了有很多方法，例如append()、pop()等等，这在后面会讲到。

要点：

- 1、List写在方括号之间，元素用逗号隔开。
- 2、和字符串一样，list可以被索引和切片。
- 3、List可以使用+操作符进行拼接。
- 4、List中的元素是可以改变的。

四、Tuple

元组（tuple）与列表类似，不同之处在于元组的元素不能修改。元组写在小括号里，元素之间用逗号隔开。元组中的元素类型也可以不相同：

```
>>> a = (1991, 2014, 'physics', 'math')
```

```
>>> print(a, type(a), len(a))
(1991, 2014, 'physics', 'math') <class 'tuple'> 4
```

元组与字符串类似，可以被索引且下标索引从0开始，也可以进行截取/切片（看上面，这里不再赘述）。其实，可以把字符串看作一种特殊的元组。

```
>>> tup = (1, 2, 3, 4, 5, 6)
>>> print(tup[0], tup[1:5])
1 (2, 3, 4, 5)
>>> tup[0] = 11 # 修改元组元素的操作是非法的
```

虽然tuple的元素不可改变，但它可以包含可变的对象，比如list列表。

构造包含0个或1个元素的tuple是个特殊的问题，所以有一些额外的语法规则：

```
tup1 = () # 空元组
tup2 = (20,) # 一个元素，需要在元素后添加逗号
```

另外，元组也支持用+操作符：

```
>>> tup1, tup2 = (1, 2, 3), (4, 5, 6)
>>> print(tup1+tup2)
(1, 2, 3, 4, 5, 6)
```

string、list和tuple都属于**sequence**（序列）。

要点：

- 1、与字符串一样，元组的元素不能修改。
- 2、元组也可以被索引和切片，方法一样。
- 3、注意构造包含0或1个元素的元组的特殊语法规则。
- 4、元组也可以使用+操作符进行拼接。

五、Sets

集合（set）是一个无序不重复元素的集。基本功能是进行成员关系测试和消除重复元素。可以使用大括号或者**set()**函数创建set集合，注意：创建一个空集合必须用**set()**而不是**{}**，因为**{}**是用来创建一个空字典。

```
>>> student = {'Tom', 'Jim', 'Mary', 'Tom', 'Jack', 'Rose'}
>>> print(student) # 重复的元素被自动去掉
{'Jim', 'Jack', 'Mary', 'Tom', 'Rose'}
>>> 'Rose' in student # membership testing（成员测试）
True
>>> # set可以进行集合运算
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'b', 'c', 'd', 'r'}
>>> a - b # a和b的差集
```



```
{'b', 'd', 'r'}
>>> a | b      # a和b的并集
{'l', 'm', 'a', 'b', 'c', 'd', 'z', 'r'}
>>> a & b      # a和b的交集
{'a', 'c'}
>>> a ^ b      # a和b中不同时存在的元素
{'l', 'm', 'b', 'd', 'z', 'r'}
```

要点：

- 1、set集合中的元素不重复，重复了它会自动去掉。
- 2、set集合可以用大括号或者set()函数创建，但空集合必须使用set()函数创建。
- 3、set集合可以用来进行成员测试、消除重复元素。

六、Dictionary

字典 (dictionary) 是Python中另一个非常有用的内置数据类型。字典是一种映射类型 (mapping type)，它是一个无序的键：值对集合。关键字必须使用不可变类型，也就是说list和包含可变类型的tuple不能做关键字。在同一个字典中，关键字还必须互不相同。

```
>>> dic = {} # 创建空字典
>>> tel = {'Jack':1557, 'Tom':1320, 'Rose':1886}
>>> tel
{'Tom': 1320, 'Jack': 1557, 'Rose': 1886}
>>> tel['Jack'] # 主要的操作：通过key查询
1557
>>> del tel['Rose'] # 删除一个键值对
>>> tel['Mary'] = 4127 # 添加一个键值对
>>> tel
{'Tom': 1320, 'Jack': 1557, 'Mary': 4127}
>>> list(tel.keys()) # 返回所有key组成的list
['Tom', 'Jack', 'Mary']
>>> sorted(tel.keys()) # 按key排序
['Jack', 'Mary', 'Tom']
>>> 'Tom' in tel      # 成员测试
True
>>> 'Mary' not in tel # 成员测试
False
```

构造函数 **dict()** 直接从键值对sequence中构建字典，当然也可以进行推导，如下：

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

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

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

另外，字典类型也有一些内置的函数，例如clear()、keys()、values()等。

要点：

- 1、字典是一种映射类型，它的元素是键值对。
- 2、字典的关键字必须为不可变类型，且不能重复。
- 3、创建空字典使用{ }。

第三篇 运算符

Python中的运算符大部分与C语言的类似，但也有很多不同的地方。本篇就大概地罗列一下Python 3中的运算符。

一、算术运算符

运算符	描述	示例
$x + y$	加	$10 + 20 = 30$
$x - y$	减	$10 - 5 = 5$
$x * y$	乘	$3 * 6 = 18$
x / y	除-返回浮点数	$2 / 4 = 0.5$
$x // y$	取整除 - 返回商的整数部分	$2 // 4 = 0$
$x \% y$	取余	$15 \% 4 = 3$
$-x$	异号	$-(-5) = 5$
$+x$	不变号	$+5 = 5$
<code>abs(x)</code>	取绝对值	<code>abs(-0.4) = 0.4</code>
<code>int(x)</code>	x转换为整数	<code>int(5.9) = 5</code>
<code>float(x)</code>	x转换为浮点数	<code>float(5) = 5.0</code>
<code>complex(re, im)</code>	返回复数 - re为实数部分 im为虚数部分	<code>complex(4,3) = 4+3j</code>
<code>c.conjugate()</code>	返回c的共轭复数	<code>c=4+3j; c.conjugate() = 4-3j</code>
<code>divmod(x, y)</code>	返回一个数值对(x/y , $x\%y$)	<code>divmod(8, 3) = (2, 2)</code>
<code>pow(x, y)</code>	x的y次幂	<code>pow(2, 5) = 32</code>
$x ** y$	x的y次幂	$2 ** 5 = 32$

Note :

1. 双斜杠 `//` 除法总是向下取整。
2. 从浮点数到整数的转换可能会舍入也可能截断，建议使用`math.floor()`和`math.ceil()`明确定义的转换。
3. Python定义`pow(0, 0)`和`0 ** 0`等于1。

二、比较运算符

运算符	描述
<	小于
<=	小于或等于
>	大于
>=	大于或等于
==	等于
!=	不等于
is	判断两个标识符是不是引用自一个对象
is not	判断两个标识符是不是引用自不同对象

Note :

1. 八个比较运算符优先级相同。
2. Python允许 `x < y <= z` 这样的链式比较，它相当于 `x < y and y <= z`。
3. 复数不能进行大小比较，只能比较是否相等。

三、逻辑运算符

运算符	描述
x or y	if x is false, then y, else x
x and y	if x is false, then x, else y
not x	if x is false, then True, else False

Note :

1. **or** 是个短路运算符，它只有在第一个运算数为False时才会计算第二个运算数的值。
2. **and** 也是个短路运算符，它只有在第一个运算数为True时才会计算第二个运算数的值。
3. **not** 的优先级比其他类型的运算符低，所以 `not a == b` 相当于 `not (a == b)`，而 `a == not b` 是错误的。

四、位运算符

运算符	描述
y x	按位或运算符
x & y	按位与运算符

$x \wedge y$	按位异或运算符
$x \ll n$	左移动运算符
$x \gg n$	右移动运算符
$\sim x$	按位取反运算符

五、赋值运算符

复合赋值运算符与算术运算符是一一对应的：

运算符	描述
<code>=</code>	简单的赋值运算符
<code>+=</code>	加法赋值运算符
<code>-=</code>	减法赋值运算符
<code>*=</code>	乘法赋值运算符
<code>/=</code>	除法赋值运算符
<code>%=</code>	取模赋值运算符
<code>**=</code>	幂赋值运算符
<code>//=</code>	取整除法运算符

六、成员运算符

Python提供了成员运算符，测试一个元素是否在一个序列（Sequence）中。

运算符	描述
<code>in</code>	如果在指定的序列中找到值返回True，否则返回False。
<code>not in</code>	如果在指定的序列中没有找到值返回True，否则返回False。

第四篇 条件与循环控制

Python的流程控制语句包括：if条件语句、while循环语句、for循环语句、range函数以及break、continue、pass控制语句。这些语句在Python中的语义和在其他语言中是一样的，所以这里就只说它们的写法。

一、if语句

if语句是最常用的条件控制语句，Python中的一般形式为：

```
if 条件一:  
    statements  
elif 条件二:  
    statements  
else:  
    statements
```

Python中用 `elif` 代替了 `else if`，所以if语句的关键字为：if - elif - else。

注意：

- 1、每个条件后面要使用冒号（:），表示接下来是满足条件后要执行的语句块。
- 2、使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。
- 3、在Python中没有switch - case语句。

示例：

```
x = int(input("Please enter an integer: "))  
if x < 0:  
    print('Negative.')elif x == 0:  
    print('Zero.')else:  
    print('Positive.')
```

二、while语句

Python中while语句的一般形式：

```
while 判断条件:  
    statements
```

同样需要注意冒号和缩进。另外，在Python中没有 `do..while` 循环。

示例：

```
a, b = 0, 1
while b < 10: # 循环输出斐波纳契数列
    print(b)
    a, b = b, a+b
```

三、for语句

Python中的for语句与C语言中的for语句有点不同：**C语言中的for语句**允许用户自定义迭代步骤和终止条件；而**Python的for语句**可以遍历任何序列（**sequence**），按照元素在序列中的出现顺序依次迭代。一般形式为：

```
for variable in sequence:
    statements
else:
    statements
```

示例：

```
words = ['cat', 'love', 'apple', 'python', 'friends']
for item in words:
    print(item, len(item))
```

如果你需要在循环体内修改你正迭代的序列，你最好是制作一个副本，这时切片标记就非常有用：

```
words = ['cat', 'love', 'apple', 'python', 'friends']
for item in words[:]: # 制作整个列表的切片副本
    if len(item) >= 6:
        words.insert(0, item)
print(words)
```

我们注意到循环语句中还可以使用 `else子句`，下面第五点有讲到。

四、range函数

如果你要遍历一个数字序列，那么内置的**range()**函数就可以派上用场了。函数**range()**常用于for循环中，用于产生一个算术数列：

```
>>> list(range(10)) # 默认从0开始
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11)) # 从1到11，前闭后开
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5)) # 5表示步长，每隔5取一个数
[0, 5, 10, 15, 20, 25]
```

示例：

```
for i in range(2, 11):  
    print(i)
```

五、break、continue、pass及else子句

break

break语句与C语言中的一样，跳出最近的for或while循环。

continue

continue语句同样是从C语言借用的，它终止当前迭代而进行循环的下次迭代。

pass

pass语句什么都不做，它只在语法上需要一条语句但程序不需要任何操作时使用。pass语句是为了保持程序结构的完整性。

else子句

在循环语句中还可以使用else子句，else子句在序列遍历结束（for语句）或循环条件为假（while语句）时执行，但循环被break终止时不执行：

```
# 循环结束执行else子句  
for i in range(2, 11):  
    print(i)  
else:  
    print('for statement is over.')
```

```
# 被break终止时不会执行else子句  
for i in range(5):  
    if(i == 4):  
        break;  
    else:  
        print(i)  
else:  
    print('for statement is over') # 不会输出
```


第五篇 函数

函数（**function**）是组织好的、可重复使用的、具有一定功能的代码段。函数能提高应用的模块性和代码的重复利用率，Python中已经提供了很多内建函数，比如`print()`，同时Python还允许用户自定义函数。

一、定义

定义函数使用关键字 `def`，后接函数名和放在圆括号()中的可选参数列表，函数内容以冒号起始并且缩进。一般格式如下：

```
def 函数名(参数列表):  
    """文档字符串"""  
    函数体  
    return [expression]
```

注意：参数列表可选，文档字符串可选，`return`语句可选。

示例：

```
def fib(n):  
    """Print a Fibonacci series"""  
    a, b = 0, 1  
    while b < n:  
        print(b, end=' ')  
        a, b = b, a+b  
    print()  
  
fib(2000) # call  
f = fib  # assignment  
f(2000)
```

函数名的值是一种用户自定义的函数类型。函数名的值可以被赋予另一个名字，使其也能作为函数使用。

二、函数变量作用域

在函数内部定义的变量拥有一个局部作用域，在函数外定义的拥有全局作用域。注意：在函数内部可以引用全局变量，但无法对其赋值（除非用 `global` 进行声明）。

```
a = 5 # 全局变量a  
  
def func1():  
    print('func1() print a =', a)
```

```
def func2():
    a = 21    # 局部变量a
    print('func2() print a =', a)

def func3():
    global a
    a = 10    # 修改全局变量a
    print('func3() print a =', a)

func1()
func2()
func3()
print('the global a =', a)
```

三、函数调用

1、普通调用

与其他语言中函数调用一样，Python中在调用函数时，需要给定和形参相同个数的实参并按顺序一一对应。

```
def fun(name, age, gender):
    print('Name:', name, 'Age:', age, 'Gender:', gender, end=' ')
    print()

fun('Jack', 20, 'man') # call
```

2、使用关键字参数调用函数

函数也可以通过 `keyword=value` 形式的关键字参数来调用，因为我们明确指出了对应关系，所以参数的顺序也就无关紧要了。

```
def fun(name, age, gender):
    print('Name:', name, 'Age:', age, 'Gender:', gender, end=' ')
    print()

fun(gender='man', name='Jack', age=20) # using keyword arguments
```

3、调用具有默认实参的函数

Python中的函数也可以给一个或多个参数指定默认值，这样在调用时可以选择性地省略该参数：

```
def fun(a, b, c=5):
    print(a+b+c)

fun(1,2)
fun(1,2,3)
```

注意：通常情况下默认值只被计算一次，但如果默认值是一个可变对象时会有所不同，如列表，字典，或

大多类的对象时。例如，下面的函数在随后的调用中会累积参数值：

```
def fun(a, L=[]):
    L.append(a)
    print(L)

fun(1) # 输出[1]
fun(2) # 输出[1, 2]
fun(3) # 输出[1, 2, 3]
```

4、调用可变参数函数

通过在形参前加一个星号（*）或两个星号（**）来指定函数可以接收任意数量的实参。

```
def fun(*args):
    print(type(args))
    print(args)

fun(1,2,3,4,5,6)

# 输出:
# <class 'tuple'>
# (1, 2, 3, 4, 5, 6)
```

```
def fun(**args):
    print(type(args))
    print(args)

fun(a=1,b=2,c=3,d=4,e=5)

# 输出:
# <class 'dict'>
# {'d': 4, 'e': 5, 'b': 2, 'c': 3, 'a': 1}
```

从两个示例的输出可以看出：当参数形如 `*args` 时，传递给函数的任意个实参会按位置被包装进一个元组（tuple）；当参数形如 `**args` 时，传递给函数的任意个 `key=value` 实参会被包装进一个字典（dict）。

5、通过解包参数调用函数

上一点说到传递任意数量的实参时会将它们打包进一个元组或字典，当然有打包也就有解包（**unpacking**）。通过单星号和双星号对List、Tuple和Dictionary进行解包：

```
def fun(a=1, b=2, c=3):
    print(a+b+c)

fun() # 正常调用
list1 = [11, 22, 33]
dict1 = {'a':40, 'b':50, 'c':60}
fun(*list1) # 解包列表
fun(**dict1) # 解包字典
```

```
# 输出:  
# 6  
# 66  
# 150
```

注：*用于解包Sequence，**用于解包字典。解包字典会得到一系列的 `key=value`，故本质上就是使用关键字参数调用函数。

四、lambda表达式

lambda关键词能创建小型匿名函数。lambda函数能接收任何数量的参数但只能返回一个表达式的值，它的一般形式如下：

```
lambda [arg1 [,arg2,.....argn]] : expression
```

lambda表达式可以在任何需要函数对象的地方使用，它们在语法上被限制为单一的表达式：

```
f = lambda x, y: x+y  
print(f(10, 20))
```

```
def make_fun(n):  
    return lambda x: x+n  
  
f = make_fun(15)  
print(f(5))
```

五、文档字符串

函数体的第一个语句可以是三引号括起来的字符串，这个字符串就是函数的文档字符串，或称为docstring。我们可以使用 `print(function.__doc__)` 输出文档：

```
def fun():  
    """Some information of this function.  
    This is documentation string."""  
    return  
  
print(fun.__doc__)
```

文档字符串主要用于描述一些关于函数的信息，让用户交互地浏览和输出。建议养成在代码中添加文档字符串的好习惯。

Python FAQ1：传值，还是传引用？

在C/C++中，传值和传引用是函数参数传递的两种方式。由于思维定式，从C/C++转过来的Python初学者也经常会感到疑惑：在Python中，函数参数传递是传值，还是传引用呢？

看下面两段代码：

```
def foo(arg):  
    arg = 5  
    print(arg)  
  
x = 1  
foo(x)    # 输出5  
print(x)  # 输出1
```

```
def foo(arg):  
    arg.append(3)  
  
x = [1, 2]  
print(x)  # 输出[1, 2]  
foo(x)  
print(x)  # 输出[1, 2, 3]
```

看完第一段代码，会有人说这是值传递，因为函数并没有改变x的值；看完第二段代码，又会有人说这是传引用，因为函数改变了x的内容。

那么，Python中的函数到底是传值还是传引用呢？看下面的解释。

一、变量和对象

我们需要先知道Python中的“变量”与C/C++中“变量”是不同的。

在C/C++中，当你初始化一个变量时，就是声明一块存储空间并写入值。相当于把一个值放入一个盒子里：

```
int a = 1;
```



现在“a”盒子里放了一个整数1，当给变量a赋另外一个值时会替换盒子a里面的内容：

```
a = 2;
```



当你把变量a赋给另外一个变量时，会拷贝a盒子中的值并放入一个新的“盒子”里：

```
int b = a;
```



在Python中，一个变量可以说是内存中的一个对象的“标签”或“引用”：

```
a = 1
```



现在变量a指向了内存中的一个int型的对象（a相当于对象的标签）。如果给a重新赋值，那么标签a将会移动并指向另一个对象：

```
a = 2
```



原来的值为1的int型对象仍然存在，但我们不能再通过a这个标识符去访问它了（当一个对象没有任何标签或引用指向它时，它就会被自动释放）。如果我们把变量a赋给另一个变量，我们只是给当前内存中对象增加一个“标签”而已：

```
b = a
```



综上所述，在Python中变量只是一个标签一个标识符，它指向内存中的对象。故变量并没有类型，类型是属于对象的，这也是Python中的变量可以被任何类型赋值的原因。

二、可变对象与不可变对象

在Python的基本数据类型中，我们知道numbers、strings和tuples是不可更改的对象，而list、dict是可

以修改的对象。那么可变与不可变有什么区别呢？看下面示例：

```
a = 1    # a指向内存中一个int型对象
a = 2    # 重新赋值
```

当将a重新赋值时，因为原来值为1的对象是不能改变的，所以a会指向一个新的int对象，其值为2。（如上面的图示）

```
lst = [1, 2] # lst指向内存中一个list类型的对象
lst[0] = 2   # 重新赋值lst中第一个元素
```

因为list类型是可以改变的，所以第一个元素变更为2。更确切的说，lst的第一个元素是int型，重新赋值时一个新的int对象被指定给第一个元素，但是对于lst来说，它所指的列表型对象没有变，只是列表的内容（其中一个元素）改变了。

好了，到这里我们就很容易解释本文开头的两段代码了：

```
def foo(arg):
    arg = 5
    print(arg)

x = 1
foo(x)    # 输出5
print(x)  # 输出1
```

上面这段代码把x作为参数传递给函数，这时x和arg都指向内存中值为1的对象。然后在函数中arg = 5时，因为int对象不可改变，于是创建一个新的int对象（值为5）并且令arg指向它。而x仍然指向原来的值为1的int对象，所以函数没有改变x变量。

```
def foo(arg):
    arg.append(3)

x = [1, 2]
print(x)    # 输出[1, 2]
foo(x)
print(x)    # 输出[1, 2, 3]
```

这段代码同样把x传递给函数foo，那么x和arg都会指向同一个list类型的对象。因为list对象是可以改变的，函数中使用append在其末尾添加了一个元素，list对象的内容发生了改变，但是x和arg仍然是指向这一个list对象，所以变量x的内容发生了改变。

那么Python中参数传递是传值，还是传引用呢？准确的回答：都不是。之所以不是传值，因为没有产生复制，而且函数拥有与调用者同样的对象。而似乎更像是C++的传引用？但是有时却不能改变实参的值。只能这样说：对于不可变的对象，它看起来像C++中的传值方式；对于可变对象，它看起来像C++中的按引用传递。

附：Everything is Object in Python

Python使用对象模型来储存数据，任何类型的值都是一个对象。所有的python对象都有3个特征：身份、类型和值。

- 身份：每一个对象都有自己的唯一的标识，可以使用内建函数**id()**来得到它。这个值可以被认为该对象的内存地址。
- 类型：对象的类型决定了该对象可以保存的什么类型的值，可以进行什么操作，以及遵循什么样的规则。**type()**函数来查看python 对象的类型。
- 值：对象表示的数据项。

```
>>> a = 1
>>> id(a)
140068196051520
>>> b = 2
>>> id(b)
140068196051552
>>> c = a
>>> id(c)
140068196051520
>>> c is a
True
>>> c is not b
True
```

运算符 `is`、`is not` 就是通过id()的返回值（即身份）来判定的，也就是看它们是不是同一个对象的“标签”。

第六篇 深入list列表

正如Python FAQ1附录中说的，Python中任何值都是一个对象，所以任何类型（int、str、list...）都是一个类。而类就必然有它的方法或属性，我们要记下这么多类的所有方法显然是不可能的，这里介绍两个小技巧：

- **dir()**：内置函数，用来查询一个类或者对象所有属性，比如 `>>> dir(list)`。
- **help()**：内置函数，用来查询具体的说明文档，比如 `>>> help(int)`。

在Python的基本数据类型中，我们初步了解了list列表，也介绍了列表是Python中使用最频繁的数据类型。本文将进一步深入学习列表的使用。

一、列表的方法

list.append(x)

在列表的尾部添加一个项，等价于 `a[len(a):] = [x]`。

list.extend(L)

将给定的列表L接到当前列表后面，等价于 `a[len(a):] = L`。

list.insert(i, x)

在给定的位置i前插入项，例如：`a.insert(0, x)` 会在列表的头部插入，而 `a.insert(len(a), x)` 则等价于 `a.append(x)`。

list.remove(x)

移除列表中第一个值为x的项，没有的话会产生一个错误。

list.pop([i])

删除列表给定位置的项，并返回它。如果没有指定索引，`a.pop()` 移除并返回列表的最后一项。（方括号表示可选）

list.clear()

删除列表中的所有项，相当于 `del a[:]`。

list.index(x)

返回列表中第一个值为x的项的索引。如果没有匹配的项，则产生一个错误。

list.count(x)

返回列表中x出现的次数。

list.sort()

就地完成列表排序。

list.reverse()

就地完成列表项的翻转。

list.copy()

返回列表的一个浅拷贝，相当于 `a[:]`。

二、列表当栈

List的方法使得其可以很方便地作为一个栈来使用。我们知道，栈的特点是最后进入的元素最先出来（即后入先出），用 `append()` 方法进行压栈，用不指定索引的 `pop()` 方法进行出栈。

示例：

```
stack = []
for x in range(1,6):
    stack.append(x)    # 入栈
    print('push', x, end=' ')
    print(stack)

print('Now stack is', stack)

while len(stack)>0:
    print('pop', stack.pop(), end=' ') # 出栈
    print(stack)
```

三、列表当队列

列表还可以当作队列来使用，队列的特性是第一个加入的元素第一个取出来（即先入先出）。然而，把列表当队列使用效率并不高，因为从列表的尾部添加和弹出元素是很快的，而在列表的开头插入或弹出是比较慢的（因为所有元素都得移动一个位置）。

要实现一个队列，使用标准库的 `collections.deque`，它被设计成在两端添加和弹出都很快。示例：

```
from collections import deque
queue = deque()    # 创建空队列
for x in range(1,6):
    queue.append(x)    # 入队
    print('push', x, end=' ')
    print(list(queue))

print('Now queue is', list(queue))

while len(queue)>0:
    print('pop', queue.popleft(), end=' ') # 出队
    print(list(queue))
```

四、列表推导式

列表推导式提供了从序列创建列表的简单途径。通常程序会对序列的每一个元素做些操作，并以其结果作为新列表的元素，或者根据指定的条件来创建子序列。

列表推导式的结构是：在一个方括号里，首先是一个表达式，随后是一个 `for` 子句，然后是零个或更多的 `for` 或 `if` 子句。返回结果是一个根据表达从其后的 `for` 和 `if` 上下文环境中生成出来的列表。 示例：

```
squares = [x**2 for x in range(10)] # 推导式
print(squares)
# 输出是[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
pairs = [(x, y) for x in [1,2,3] for y in [3,1,4] if x!=y] # 推导式
print(pairs)
# 输出是[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

五、列表嵌套

Python中并没有二维数组的概念，但我们可以通过列表嵌套达到同样的目的。

```
mat = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

同样，我们可以使用推导式生成嵌套的列表：

```
mat = [[1,2,3], [4,5,6], [7,8,9]]
new_mat = [ [row[i] for row in mat] for i in [0,1,2] ] # 嵌套
print(new_mat)
```

附：del语句

`del` 语句可以通过给定索引（而不是值）来删除列表中的项，它与返回一个值的`pop()`方法不同。`del`语句也可以移除列表中的切片，或者清除整个列表：

```
lst = [1,2,3,4,5,6,7,8,9]
del lst[2] # 删除指定索引项
print(lst)
del lst[2:5] # 删除切片
print(lst)
del lst[:] # 删除整个列表
print(lst)
```

`del`也可以用于删除变量实体：

```
del lst
```

在删除变量实体之后引用 `lst` 的话会产生错误。

第七篇 输入和输出

一个程序可以从键盘读取输入，也可以从文件读取输入；而程序的结果可以输出到屏幕上，也可以保存到文件中便于以后使用。本文介绍Python中最基本的I/O函数。

一、控制台I/O

读取键盘输入

内置函数 `input([prompt])`，用于从标准输入读取一个行，并返回一个字符串（去掉结尾的换行符）：

```
s = input("Enter your input:")
```

注：在Python 3.x版本中取消了 `raw_input()` 函数。

打印到屏幕

最简单的输出方法是用`print`语句，你可以给它传递零个或多个用逗号隔开的表达式：

```
print([object, ...][, sep=' '][, end='newline_character_here'[, file=redirect_to_here]])
```

方括号内是可选的，`sep`表示分割符，`end`表示结束符，`file`表示重定向文件。如果要给`sep`、`end`、`file`指定值必须使用关键字参数。

```
print('hello', 'world', sep='%')    # 输出hello%world
print('hello', 'world', end='*')    # 输出hello world*，并且不换行
```

二、文件I/O

读写文件之前，先用`open()`函数打开一个文件，它会返回一个文件对象（file object）：

```
f = open(filename, mode)
```

如果不指定`mode`参数，文件将默认以‘r’模式打开。模式中的字符有：

- **r**：只读
- **w**：只写，如果文件已存在则将其覆盖。如果该文件不存在，创建新文件
- **+**：读写（不能单独使用）
- **a**：打开文件用于追加，只写，不存在则创建新文件
- **b**：以二进制模式打开（不能单独使用）

所以可能的模式大概有**r**、**w**、**r+**、**w+**、**rb**、**wb**、**rb+**、**wb+**、**a**、**a+**、**ab**、**ab+**，注意只有**w**和**a**可以创建文件。

通常情况下，文件都是以文本模式（text mode）打开的，也就是说，从文件中读写的是以一种特定的编码格式进行编码（默认的是 UTF-8）的字符串。如果文件以二进制模式（binary mode）打开，数据将以字节对象的形式进行读写：

```
f = open('a.txt', 'wb+')
f.write('I like apple!') # 报错
f.write(b'I like apple!') # 以bytes对象的形式进行读写
```

Bytes对象是0到127的不可修改的整数序列，或纯粹的 ASCII 字符，它的用途是存储二进制数据。

1. 可以通过在一个字符串前面加上 **'b'** 来创建一个 bytes literal；
2. 也可以通过 **bytes()** 函数创建一个 bytes 对象。

注意：如果 bytes() 函数的初始化器是一个字符串，那么必须提供一种编码。

```
b1 = b'This is string'
b2 = bytes('This is string', 'UTF-8') # 必须指定编码格式
```

字符串对象与字节对象是不兼容的，要将 bytes 转变为 str，bytes 对象必须要进行解码，使用 **decode()** 方法：

```
b = bytes('This is string', 'UTF-8')
print(b, b.decode(), sep='\n')
# 输出:
# b'This is string'
# This is string
```

文件对象的方法（假设 f 是一个文件对象）：

- **f.read(size)**：读取 size 个字节的数据，然后作为字符串或 bytes 对象返回。size 是一个可选参数，如果不指定 size，则读取文件的所有内容。
- **f.readline()**：读取一行。在字符串末尾会留下换行符 (\n)，如果到文件尾，返回空字符串。
- **f.readlines()**：读取所有行，储存在列表中，每个元素是一行，相当于 `list(f)`。
- **f.write(string)**：将 string 写入到文件中，返回写入的字符数。如果以二进制模式写文件，需要将 string 转换为 bytes 对象。
- **f.tell()**：返回文件对象当前所处的位置，它是从文件开头开始算起的字节数。
- **f.seek(offset, from_what)**：改变文件对象所处的位置。offset 是相对参考位置的偏移量，from_what 取值 0（文件头，默认）、1（当前位置）、2（文件尾）表示参考位置。
- **f.close()**：关闭文件对象。

这些都是很常用的方法，当然文件对象不止这些方法。根据打开的模式不同，open() 返回的文件对象类型也不同：

- **TextIOWrapper**：文本模式，返回 TextIOWrapper 对象。
- **BufferedReader**：读二进制，即 rb，返回 BufferedReader 对象。
- **BufferedWriter**：写和追加二进制，即 wb、ab，返回 BufferedWriter 对象。
- **BufferedRandom**：读/写模式，即含有 + 的模式，返回 BufferedRandom 对象。

可以在这些文件对象上运行 dir() 或 help()，查看它们所有的方法。

补充：

- 1、在文本模式下，`seek()`方法只会相对于文件起始位置进行定位。（除了定位文件尾可以用 `seek(0, 2)` 之外）
- 2、可以循环迭代一个文件对象一行一行读取：

```
for line in f:
    print(line, end='')
```

三、格式化输出

一般来说，我们希望更多的控制输出格式，而不是简单的以空格分割。这里有两种方式：

- 第一种是由你自己控制。使用字符串切片、连接操作以及 `string` 包含的一些有用的操作。
- 第二种是使用 `str.format()` 方法。

下面给一个示例：

```
# 第一种方式：自己控制
for x in range(1, 11):
    print(str(x).rjust(2), str(x*x).rjust(3), end=' ')
    print(str(x*x*x).rjust(4))

# 第二种方式：str.format()
for x in range(1, 11):
    print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))

# 输出都是：
# 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
```

第一种方式中，字符串对象的 `str.rjust()` 方法的作用是将字符串靠右，并默认在左边填充空格，类似的方法还有 `str.ljust()` 和 `str.center()`。这些方法并不会写任何东西，它们仅仅返回新的字符串，如果输入很长，它们并不会截断字符串。我们注意到，同样是输出一个平方与立方表，使用 `str.format()` 会方便很多。

`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!"
```

括号及括号里的字符将会被 format() 中的参数替换。括号中的数字用于指定传入对象的位置：

```
>>> print('{0} and {1}'.format('Kobe', 'James'))
Kobe and James
>>> print('{1} and {0}'.format('Kobe', 'James'))
James and Kobe
```

如果在 format() 中使用了关键字参数，那么它们的值会指向使用该名字的参数：

```
>>> print('The {thing} is {adj}.'.format(thing='flower', adj='beautiful'))
The flower is beautiful.
```

可选项 ':' 和格式标识符可以跟着 field name，这样可以进行更好的格式化：

```
>>> import math
>>> print('The value of PI is {0:.3f}'.format(math.pi))
The value of PI is 3.142.
```

在 ':' 后传入一个整数，可以保证该域至少有这么多的宽度，用于美化表格时很有用：

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

我们还可以将参数解包进行格式化输出。例如，将table解包为关键字参数：

```
table = {'Jack':4127, 'Rose':4098, 'Peter':7678}
print('Jack is {Jack}, Rose is {Rose}, Peter is {Peter}.'.format(**table))
# 输出: Jack is 4127, Rose is 4098, Peter is 7678.
```

补充：

% 操作符也可以实现字符串格式化。它将左边的参数作为类似 sprintf() 式的格式化字符串，而将右边的代入：

```
import math
print('The value of PI is %10.3f.' %math.pi)
# 输出: The value of PI is      3.142.
```

因为这种旧式的格式化最终会从Python语言中移除，应该更多的使用 str.format()。

附：文本模式与二进制模式

1、在Windows系统中，文本模式下，默认是将Windows平台的行末标识符 `\r\n` 在读时转为 `\n`，而在写时将 `\n` 转为 `\r\n`。这种隐藏的行为对于文本文件是没有问题的，但是对于二进制数据像 JPEG 或 EXE 是会出问题的。在使用这些文件时请小心使用二进制模式。

2、在类Unix/Linux系统中，行末标识符为 `\n`，即文件以 `\n` 代表换行。所以Unix/Linux系统中在文本模式和二进制模式下并无区别。

Python FAQ2：赋值、浅拷贝、深拷贝的区别？

在写Python过程中，经常会遇到对象的拷贝，如果不理解浅拷贝和深拷贝的概念，你的代码就可能出现一些问题。所以，在这里按个人的理解谈谈它们之间的区别。

一、赋值（assignment）

在《Python FAQ1》一文中，对赋值已经讲的很清楚了，关键要理解变量与对象的关系。

```
>>> a = [1, 2, 3]
>>> b = a
>>> print(id(a), id(b), sep='\n')
139701469405552
139701469405552
```

在Python中，用一个变量给另一个变量赋值，其实就是给当前内存中的对象增加一个“标签”而已。

如上例，通过使用内置函数 `id()`，可以看出 `a` 和 `b` 指向内存中同一个对象。`a is b` 会返回 `True`。

二、浅拷贝（shallow copy）

注意：浅拷贝和深拷贝的不同仅仅是对组合对象来说，所谓的组合对象就是包含了其它对象的对象，如列表，类实例。而对于数字、字符串以及其它“原子”类型，没有拷贝一说，产生的都是原对象的引用。

所谓“浅拷贝”，是指创建一个新的对象，其内容是原对象中元素的引用。（拷贝组合对象，不拷贝子对象）

常见的浅拷贝有：切片操作、工厂函数、对象的`copy()`方法、`copy`模块中的`copy`函数。

```
>>> a = [1, 2, 3]
>>> b = list(a)
>>> print(id(a), id(b))           # a和b身份不同
140601785066200 140601784764968
>>> for x, y in zip(a, b):       # 但它们包含的子对象身份相同
...     print(id(x), id(y))
...
140601911441984 140601911441984
140601911442016 140601911442016
140601911442048 140601911442048
```

从上面可以明显的看出来，`a` 浅拷贝得到 `b`，`a` 和 `b` 指向内存中不同的 `list` 对象，但它们的元素却指向

相同的 **int** 对象。这就是浅拷贝！

三、深拷贝 (**deep copy**)

所谓“深拷贝”，是指创建一个新的对象，然后递归的拷贝原对象所包含的子对象。深拷贝出来的对象与原对象没有任何关联。

深拷贝只有一种方式：copy模块中的deepcopy函数。

```
>>> import copy
>>> a = [1, 2, 3]
>>> b = copy.deepcopy(a)
>>> print(id(a), id(b))
140601785065840 140601785066200
>>> for x, y in zip(a, b):
...     print(id(x), id(y))
...
140601911441984 140601911441984
140601911442016 140601911442016
140601911442048 140601911442048
```

看了上面的例子，有人可能会疑惑：

为什么使用了深拷贝，**a**和**b**中元素的**id**还是一样呢？

答：这是因为对于不可变对象，当需要一个新的对象时，python可能会返回已经存在的某个类型和值都一致的对象的引用。而且这种机制并不会影响 **a** 和 **b** 的相互独立性，因为当两个元素指向同一个不可变对象时，对其中一个赋值不会影响另外一个。

我们可以用一个包含可变对象的列表来确切地展示“浅拷贝”与“深拷贝”的区别：

```
>>> import copy
>>> a = [[1, 2],[5, 6], [8, 9]]
>>> b = copy.copy(a)           # 浅拷贝得到b
>>> c = copy.deepcopy(a)       # 深拷贝得到c
>>> print(id(a), id(b))        # a 和 b 不同
139832578518984 139832578335520
>>> for x, y in zip(a, b):     # a 和 b 的子对象相同
...     print(id(x), id(y))
...
139832578622816 139832578622816
139832578622672 139832578622672
139832578623104 139832578623104
>>> print(id(a), id(c))       # a 和 c 不同
139832578518984 139832578622456
>>> for x, y in zip(a, c):     # a 和 c 的子对象也不同
...     print(id(x), id(y))
...
139832578622816 139832578621520
139832578622672 139832578518912
139832578623104 139832578623392
```

从这个例子中可以清晰地看出浅拷贝与深拷贝地区别。

总结：

- 1、赋值：简单地拷贝对象的引用，两个对象的id相同。
- 2、浅拷贝：创建一个新的组合对象，这个新对象与原对象共享内存中的子对象。
- 3、深拷贝：创建一个新的组合对象，同时递归地拷贝所有子对象，新的组合对象与原对象没有任何关联。虽然实际上会共享不可变的子对象，但不影响它们的相互独立性。

浅拷贝和深拷贝的不同仅仅是对组合对象来说，所谓的组合对象就是包含了其它对象的对象，如列表，类实例。而对于数字、字符串以及其它“原子”类型，没有拷贝一说，产生的都是原对象的引用。

第八篇 模块

在程序中定义函数可以实现代码重用。但当你的代码逐渐变得庞大时，你可能想要把它分割成几个文件，以便能够更简单地维护。同时，你希望在一个文件中写的代码能够被其他文件所重用，这时我们应该使用模块（**module**）。

一、导入模块

在Python中，一个 `.py` 文件就构成一个模块。一个模块中的定义可以导入（**import**）到另一个模块或主模块。

比如你可以通过内置模块platform来查看你当前的操作平台信息：

```
import platform
s = platform.platform()
print(s)
# 我的输出: Linux-3.15.8-200.fc20.x86_64-x86_64-with-fedora-20-Heisenbug
```

又比如你可以通过内置模块time获取当前的时间：

```
import time
s = time.ctime()
print(s)
# 输出: Mon Aug 18 16:04:57 2014
```

每个模块有其私有的符号表，在该模块内部当成全局符号表来使用。 当我们将一个模块导入到当前模块时，只有被导入模块的名称被放入当前模块的全局符号表里，所以我们不用担心变量名发生冲突。

其他几种导入方式：

1、 `import a as b` ：导入模块a，并将模块a重命名为b。

```
import time as x
s = x.ctime()
print(s)
```

2、 `from a import func` ：直接把模块内的函数或变量的名称导入当前模块符号表里。

```
from time import ctime
s = ctime() # 这时可以直接调用函数，而不用再使用time.ctime()
print(s)
```

3、 `from a import *` ：导入模块中所有的名字（以下划线开头的名字除外）到当前模块符号表里。

```
from time import *
s = ctime()
print(s)
```

注意：导入 * 是不好的，因为它常常产生难以阅读的代码，并且会容易产生名字冲突。

二、模块搜索路径

当导入名为 `a` 的模块时，解释器会先从内建模块尝试匹配，如果没找到，则将在 `sys.path` 记录的所有目录中搜索 `a.py` 文件，而 `sys.path` 则包括：

- 当前程序所在目录
- 标准库的安装目录
- 操作系统环境变量 `PYTHONPATH` 所包含的目录

变量 `sys.path` 是一个字符串列表，它为解释器指定了模块的搜索路径。它通过环境变量 `PATHONPATH` 初始化为一个默认路径，当没有设置 `PYTHONPATH` 时，就使用内建默认值来初始化。你可以通过标准 `list` 操作来修改它：

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

三、模块的 `__name__`

对于任何一个模块，模块的名字都可以通过全局变量 `__name__` 得到：

```
import time
s = time.__name__
print(s)    # 输出time
```

一个模块既可以在其它模块中导入使用，也可以当作脚本直接运行。不同的是，当导入到其他模块时，`__name__` 的值是被导入模块的名字；而当作为脚本运行时，`__name__` 的值被设为“`__main__`”：

```
# test.py
if __name__ == '__main__':
    print('This program is being run by itself')
else:
    print('I am being imported into another module')
```

当作脚本执行：

```
$ python test.py
This program is being run by itself
```

当作导入模块使用：

```
>>> import test
I am being imported into another module
>>>
```

四、dir() 函数

在Python3基础六中我们提到，可以通过内置dir()函数查询一个类或者对象的所有属性。除此之外，我们还可以用它列出一个模块里定义的所有名字，它返回一个有序字符串列表：

```
>>> import builtins
>>> dir(builtins)
```

五、包

可以把多个模块，即多个.py文件，放在同一个文件夹中，构成一个包（**Package**）。例如：

sound/	顶级包
__init__.py	初始化这个声音包
formats/	格式转换子包
__init__.py	
wavread.py	
wavwrite.py	
aiffread.py	
aiffwrite.py	
auread.py	
auwrite.py	
...	
effects/	音效子包
__init__.py	
echo.py	
surround.py	
reverse.py	
...	
filters/	过滤器子包
__init__.py	
equalizer.py	
vocoder.py	
karaoke.py	
...	

注意：在每个包文件夹里都必须包含一个 __init__.py 的文件，告诉Python，该文件夹是一个包。__init__.py 可以是一个空文件。

我们可以通过 `import 包名.模块名` 导入包中的子模块，例如：

```
import sound.effects.echo
```

当然，也可以使用 `from... import...` 句式导入包中的模块：

from sound.effects import echo	# 导入echo子模块
from sound.effects.echo import echofilter	# 导入echo子模块中的函数或变量
from sound.effects import *	# 导入__all__变量中所有的子模块

第九篇 错误和异常

本文主要介绍Python中的错误和异常，涉及到简单的异常处理、抛出异常以及清理动作。至于自定义异常类，将在介绍类与继承的时候讲到。

一、定义

常见的两种错误：语法错误 和 异常。

1、语法错误 (Syntax Errors)

语法错误，也就是解析时错误。当我们写出不符合python语法的代码时，在解析时会报SyntaxError，并且会显示出错的那一行，并用小箭头指明最早探测到错误的位置。比如：

```
x = input('please input an integer:')
if int(x) > 5:
    print 'hello world'
```

在python 3中会报语法错误：

```
File "/home/songlee/test", line 3
    print 'hello world'
          ^
SyntaxError: invalid syntax
```

2、异常 (Exceptions)

即使语句或表达式在语法上是正确的，但在尝试运行时也可能发生错误，运行时错误就叫做 异常 (Exceptions)。异常并不是致命的问题，因为我们可以对异常进行处理。

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> 2 + x*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

上面展示了三种exception的类型：ZeroDivisionError、NameError、TypeError，它们都是内置异常的名称。标准异常的名字是内建的标识符 (但并不是关键字)。

二、处理异常 (**try...except...**)

我们可以使用 `try...except...` 语句来处理异常。`try` 语句块中是要执行的语句，`except` 语句块中是异常处理语句。一个 `try` 语句可以有多条的 `except` 语句，用以指定不同的异常，但至多只有一个会被执行：

```
try:
    x = int(input('please input an integer:'))
    if 30/x > 5:
        print('Hello World!')
except ValueError:
    print('That was no valid number. Try again...')
except ZeroDivisionError:
    print('The divisor can not be zero, Try again...')
except:
    print('Handling other exceptions...')
```

上面这段代码，当输入a（非数字）时，将抛出**`ValueError`**异常；当输入0时，将抛出**`ZeroDivisionError`**异常；当抛出其他类型的异常时，将执行`except:`后的处理语句。

如果在 `try` 语句执行时，出现了一个异常，该语句的剩下部分将被跳过。并且如果该异常的类型匹配到了 `except` 后面的异常名，那么该 `except` 后的语句将被执行。注意，如果 `except` 后面没有跟异常名，表示它匹配任何类型的异常，`except:` 必须放在最后。

一个 `except` 语句可以同时包括多个异常名，但需要用括号括起来，比如：

```
except (RuntimeError, TypeError, NameError):
    pass
```

`try / except` 语句可以有一个可选的 `else` 语句。`else` 语句必须要放在所有 `except` 语句后面，当没有异常发生的时候，`else` 从句将被执行：

```
try:
    name = input('please input an integer:')
    f = open(name, 'r')
except IOError:
    print('Cannot open', name)
except:
    print('Unexpected errors.')
else:
    print('close the file', name)
    f.close()
```

三、抛出异常 (**raise**)

`raise` 语句允许程序员强制地抛出一个特定的异常，例如：

```
>>> raise NameError('HiThere')      # 抛出异常
```



```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

raise 抛出的异常必须是一个异常实例或类（派生自 Exception 的类）。

四、清理动作（finally）

try 语句有另一种可选的 `finally` 从句，用于自定义一些扫尾清理的工作。

```
try:
    x = int(input('please input an integer:'))
    if x > 5:
        print('Hello World!')
except ValueError:
    print('It was not a number. Try again.')
finally:
    print('Some clean-up actions!')
```

与 `else` 从句的区别在于：`else` 语句只在没有异常发生的情况下执行，而 `finally` 语句则不管异常发生与否都会执行。准确的说，`finally` 语句总是在退出 `try` 语句前被执行，无论是正常退出、异常退出，还是通过 `break`、`continue`、`return` 退出。

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print('error: division by zero!')
...     else:
...         print('executing else-clause,', 'result is', result)
...     finally:
...         print('executing finally-clause')
...

>>> divide(2, 1)      # 正常退出
executing else-clause, result is 2.0
executing finally-clause

>>> divide(2, 0)      # 异常退出
error: division by zero!
executing finally-clause

>>> divide('2', '1')  # 异常退出，异常未被处理。
executing finally-clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

从上面看出，`finally` 语句在任何情况下都被执行了。对于没有被 `except` 处理的异常，将在执行完 `finally` 后被重新抛出。

另外，有些对象预定义了标准的清理动作（**clean-up actions**）。当对象不再需要时，该动作将被执行，无论对其使用的操作是否成功。例如下面的[文件I/O](#)例子：

```
for line in open("myfile.txt"):
    print(line, end="")
```

这段代码的问题在于，在此代码成功执行后，文件依然被打开着。但 `with` 语句可以让文件对象在使用后被正常的清理掉：

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

在执行该语句后，文件 `f` 就会被关闭，就算是在读取时碰到了问题，文件 `f` 也会被关闭。像文件这样的对象，总会提供预定义的清理工作。

第十篇 类的初印象

Python是一种面向对象的脚本语言，所以它也提供了面向对象编程的所有基本特征：允许多继承的类继承机制、派生类可以重写它父类的任何方法、一个方法可以调用父类中同名的方法、对象可以包含任意数量和类型的数据成员。关于继承，将在下一篇博文里面介绍，本文只简单的介绍Python中的类的定义和使用。

一、类定义

最简单的类的定义形式：

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

类定义会创建一个新的命名空间，作为一个局部的作用域。在Python中，类本身就是对象，当一个类定义结束后，一个 Class Object 就被创建。

二、类对象

类对象 (Class Object) 支持两种操作：属性引用 和 实例化。

属性引用

类对象的属性引用和 Python 中所有的属性引用一样，形式为：`obj.name`。类对象创建后，类命名空间中所有的名字都是有效属性名，像下面这个类：

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

它有一个属性 `i` 和 方法 `f`，所以可以用 `MyClass.i` 和 `MyClass.f` 进行属性引用，分别返回一个整数和一个函数对象。`__doc__` 也是一个合法的属性，返回属于这个类的文档字符串。

实例化

类的实例化形式为：

```
x = MyClass()
```

创建了一个新的实例，并且将其指定给局部变量 `x`。

在创建实例时，通常可能都需要有特定的初始状态，所以一个类可以定义一个名为 `__init__()` 的特殊方法（构造方法）：

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

当一个类定义了 `__init__()` 方法，类在实例化时会自动调用 `__init__()` 方法，用于创建新的类实例。就像 C++ 中的构造函数一样，`__init__()` 也可以有更多的参数，这时实例化提供给类的参数会传给 `__init__()`，比如：

```
class student:  
    def __init__(self, n, a):  
        self.name = n  
        self.age = a  
  
stu = student('Selena', 19)  
print(stu.name, stu.age)    # 输出: Selena 19
```

三、实例对象

类对象实例化得到实例对象（**Instance Object**），实例对象只能进行 属性引用 这一种操作。合法的属性有两种：数据属性 和 方法。

数据属性

数据属性（data attributes）相当于 C++ 中的数据成员，在 Python 中，数据属性不需要声明，当它们第一次指定时就会被引入：

```
class MyClass:  
    i = 12345  
    def f(self):  
        return 'hello world'  
  
x = MyClass()  
x.counter = 1  
print(x.counter)  
del x.counter
```

注：在 Python 中每个值都是一个对象，可以通过 `object.__class__` 来获取对象的 `class`（即类型），其作用与 `type()` 相同。

方法

在类对象中定义的函数与普通函数只有一个特别的区别：它们的第一个参数必须是 `self`，用以指定调用该方法的实例对象。

注意：类的方法只有被绑定到实例对象上才能够被调用。比如上面的例子中，`x` 是 `MyClass` 类的一个实例对象，所以它可以直接调用 `f` 方法：

```
x.f()
```

为什么 `f()` 定义时是有一个参数的，而这里调用不需要参数呢？因为在调用时，`x` 对象作为参数传递给了函数的第一个参数（即 `self`）。也就是说，`x.f()` 是严格等价于 `MyClass.f(x)` 的。

所以在多数情况下，调用一个方法（有个 `n` 个参数），和调用相应的函数（也有那 `n` 个参数，但是再额外加入一个使用该方法的对象）是等价的。

另外，函数也可以在 `class` 外定义，指定该函数对象给类中的局部变量就可以了，例如：

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, y)

class C:
    f = f1
    def g(self):
        return 'hello world'

c = C()    # 实例化
c.f(1,3)
c.g()
```

四、私有成员

从C++术语上讲，Python 类的成员（包括数据成员）通常都是 **public** 的，并且所有的成员函数都是 **virtual** 的。

那么，如何在类中定义私有变量或私有方法呢？

答：在Python中规定，以两个下划线开头的名字为私有成员，不能在类的外部使用。

示例：

```
class A:
    __str = 'python'
    def __f(self):
        return self.__str
    def f(self):
        return self.__str

a = A()
a.__str    # 'A' object has no attribute '__str'
a.__f()    # 'A' object has no attribute '__f'
a.f()      # 输出: python
```

附：作用域的探讨

在讲函数变量作用域时，曾经说过在一个局部作用域内重新绑定全局变量，需要使用 `global` 声明。否则，尝试给这个变量赋值，只是会简单的创建一个新的局部变量，而不会改变那个全局变量。

这里再介绍一个 `nonlocal` 语句，它用于指示，在外层的局部作用域中的变量可以在这里进行重新绑定。下面是一个例子：

```
def scope_test():    # 作用域测试
    def do_local():
        x = 'local x'
    def do_nonlocal():
        nonlocal x
        x = 'nonlocal x'
    def do_global():
        global x
        x = 'global x'

    x = 'test x'    # 局部变量

    do_local()
    print('After do_local():', x)
    do_nonlocal()
    print('After do_nonlocal():', x)
    do_global()
    print('After do_global():', x)

scope_test()
print('In global scope:', x)
```

可以看出，局部的赋值 `do_local()` 并没有改变 `scope_test` 绑定的 `x` 变量，而 `do_nonlocal()` 则改变了 `scope_test` 中的 `x`，而 `do_global()` 则改变了模块级别的绑定，即全局变量。

第十一篇 类的拓展

在类的初印象中，我们已经简单的介绍了类，包括类的定义、类对象和实例对象。本文将进一步学习类的继承、迭代器、发生器等等。

一、类的继承

单继承

派生类的定义如下：

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

基类名 BaseClassName 对于派生类来说必须是可见的。也可以继承在其他模块中定义的基类：

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

对于派生类的属性引用：首先会在当前的派生类中搜索，如果没有找到，则会递归地去基类中寻找。

从C++术语上讲，Python 类中所有的方法都是 `virtual` 的，所以派生类可以覆写（`override`）基类的方法。在派生类中一个覆写的方法可能需要调用基类的方法，可以通过以下方式：

```
BaseClassName.method(self, arguments)
```

介绍两个函数：

- `isinstance(object, class_name)`：内置函数，用于判断实例对象 `object` 是不是类 `classname` 或其派生类的实例，即 `object.__class__` 是 `class_name` 或其派生类时返回 `True`。
- `issubclass(class1, class2)`：内置函数，用于检查类 `class1` 是不是 `class2` 的派生类。例如 `issubclass(bool, int)` 会返回 `True`，因为 `bool` 是 `int` 的派生类。

多重继承

Python支持多重继承，一个多重继承的定义形如：

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

大多数的情况（未使用`super`）下，多重继承中属性搜索的方式是，深度优先，从左到右。在继承体系中，同样的类只会被搜寻一次。如果一个属性在当前类中没有被找到，它就会搜寻 `Base1`，然后递归地搜寻 `Base1` 的基类，然后如果还是没有找到，那么就会搜索 `Base2`，依次类推。

对于菱形继承，Python 3采用了 C3 线性化算法去搜索基类，保证每个基类只搜寻一次。所以对于使用者，无须担心这个问题，如果你想了解更多细节，可以看看[Python类的方法解析顺序](#)。

二、自定义异常类

在《[Python3的错误和异常](#)》中，我们简单地介绍了Python中的异常处理、异常抛出以及清理动作。在学习了类的继承以后，我们就可以定义自己的异常类了。

自定义异常需要从 **Exception** 类派生，既可以是直接也可以是间接。例如：

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)

try:
    raise MyError(2*2)
except MyError as e:
    print('My exception occurred, value:', e.value)
# 输出: My exception occurred, value: 4
```

在这个例子中，`Exception` 的默认方法 `__init__()` 被覆写了，现在新的异常类可以像其他的类一样做任何的事。当创建一个模块时，可能会有多种不同的异常，一种常用的做法就是，创建一个基类，然后派生出各种不同的异常：

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

需要特别注意的是，如果一个 `except` 后跟了一个异常类，则这个 `except` 语句不能捕获该异常类的基类，但能够捕获该异常类的子类。例如：


```

class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass

for e in [B, C, D]:
    try:
        raise e()
    except D:
        print('D')
    except C:
        print('C')
    except B:
        print('B')

```

上面的代码会按顺序输出B、C、D。如果将三个 except 语句逆序，则会打印B、B、B。

三、迭代器 (Iterator)

到目前为止，你可能注意到，大多数的容器对象都可以使用 for 来迭代：

```

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)

```

这种形式可以说是简洁明了。其实，for 语句在遍历容器的过程中隐式地调用了 `iter()`，这个函数返回一个迭代器对象，迭代器对象定义了 `__next__()` 方法，用以在每次访问时得到一个元素。当没有任何元素时，`__next__()` 将产生 `StopIteration` 异常来告诉 for 语句停止迭代。

内置函数 `next()` 可以用来调用 `__next__()` 方法，示例：

```

>>> s = 'abc'
>>> it = iter(s)    # 获取迭代器对象
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>

```

StopIteration

在了解了迭代器的机制之后，就可以很简单的将迭代行为增加到你的类中。定义一个 `__iter__()` 方法返回一个具有 `__next__()` 的对象，如果这个类定义了 `__next__()`，那么 `__iter__()` 仅需要返回 `self`：

```
class Reverse:
    """ 逆序迭代一个序列 """
    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 -= 1
        return self.data[self.index]
```

测试：

```
# 测试
rev = Reverse('spam')
for c in rev:
    print(c, end=' ') # 输出: m a p s

# 单步测试
>>> rev = Reverse('spam')
>>> it = iter(rev) # 返回的 self 本身
>>> next(it) # 相当于 next(rev)，因为iter(rev)返回本身
'm'
>>> next(it)
'a'
>>> next(it)
'p'
>>> next(it)
's'
```

四、生成器 (Generator)

生成器 (Generator) 是用来创建迭代器的工具，它的形式跟函数一样，唯一的不同的是生成器使用 `yield` 语句返回，而不是 `return` 语句。

有了生成器，我们不再需要自定义迭代器类（例如上面的 `class Reverse`），因为自定义迭代器类需要手动实现 `__iter__()` 和 `__next__()` 方法，也是有点麻烦。而生成器则会自动创建 `__iter__()` 和 `__next__()`，可以更方便地生成一个迭代器，而且代码也会更短更简洁。例如，这里用生成器实现与 `class Reverse` 相同作用的迭代器：

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

原来要十多行代码写一个迭代器类，现在使用生成器只需要3行代码！来测试一下：

```
# 测试
for c in Reverse('spam'):
    print(c, end=' ')    # 输出: m a p s

# 单步测试
>>> rev = Reverse('spam')
>>> next(rev)
'm'
>>> next(rev)
'a'
>>> next(rev)
'p'
>>> next(rev)
's'
```

怎么样？现在感受到生成器的强大了吧。确实，生成器让我们可以方便的创建迭代器，而不必去自定义迭代器类那么麻烦。下面我们来了解一下生成器的工作过程：

```
def generator_func():
    """ 这是一个简单的生成器 """
    yield 1
    yield 2
    yield 3

# 测试
>>> g = generator_func()
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

执行过程大致如下：

1. 调用生成器函数将返回一个生成器。
2. 第一次调用生成器的 `next` 方法时，生成器才开始执行生成器函数。直到遇到 `yield` 时暂停执行（挂起），并且将 `yield` 的参数作为此次的返回值。
3. 之后每次调用 `next` 方法，生成器将从上次暂停的位置恢复并继续执行，直到再次遇到 `yield` 时暂停，同样将 `yield` 的参数返回。
4. 当调用 `next` 方法时生成器函数结束，则此次调用将抛出 `StopIteration` 异常（for循环终止条件）。

所以说，生成器的神奇之处在于每次使用 `next()` 执行生成器函数遇到 `yield` 返回时，生成器函数的“状态”会被冻结，所有的数据值和执行位置会被记住，一旦 `next()` 再次被调用，生成器函数会从它上次离开的地方继续执行。

五、类用作ADT

有些时候，类似于 Pascal 的“record”或 C 的“struct”这样的数据类型非常有用，绑定一些命名的数据。在 Python 中一个空的类定义就可以做到：

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

一段 Python 代码中如果需要一个抽象数据类型，那么可以通过传递一个类给那个方法，就好像有了那个数据类型一样。

例如，如果你有一个函数用于格式化某些从文件对象中读取的数据，你可以定义一个有 `read()` 和 `readline()` 方法的类用于读取数据，然后将这个类作为一个参数传递给那个函数。

附：类变量与实例变量的区别

类变量（**class variable**）是类的属性和方法，它们会被类的所有实例共享。而实例变量（**instance variable**）是实例对象所特有的数据。如下：

```
class animal:
    kind = 'dog' # class variable shared by all instances

    def __init__(self, color):
        self.color = color # instance variable unique to each instance

a1 = animal('black')
a2 = animal('white')

print(a1.kind, a2.kind) # shared by all animals
print(a1.color, a2.color) # unique to each animal
```

当类变量（被所有实例共享）是一个可变的对象时，如 `list`、`dict`，那么在一个实例对象中改变该属性，其他实例的这个属性也会发生变化。这应该不难理解，例如：

```

class animal:
    actions = []    # class variable shared by all instances

    def __init__(self, color):
        self.color = color    # instance variable unique to each instance

    def addActions(self, action):
        self.actions.append(action)

a1 = animal('black')
a2 = animal('white')

a1.addActions('run')    # 动物a1会跑
a2.addActions('fly')    # 动物a2会飞

print(a1.actions, a2.actions)    # 输出: ['run', 'fly'] ['run', 'fly']

```

输出结果显示：动物 a1 和 a2 总是有相同的行为（actions），显然这不是我们想要的，因为不同的动物有不同的行为，比如狗会跑、鸟会飞、鱼会游.....

对这个问题进行改进，我们只需要将 **actions** 这个属性变成实例变量，让它对每个实例对象都 unique，而不是被所有实例共享：

```

class animal:

    def __init__(self, color):
        self.color = color    # instance variable
        self.actions = []    # instance variable

    def addActions(self, action):
        self.actions.append(action)

a1 = animal('black')
a2 = animal('white')

a1.addActions('run')    # 动物a1会跑
a2.addActions('fly')    # 动物a2会飞

print(a1.actions, a2.actions)    # 输出: ['run'] ['fly']

```

(完)