Python 基础编程

第 1 章 基础语法

1. Python 中默认的编码格式是 ASCII 格式，在没修改编码格式时无法正确打印汉字，所以在读取中文时会报错。解决方法为只要在文件第一行加入#coding=utf-8 就行了。

```
#coding=utf-8
#!/usr/bin/python
print "你好，世界";
```

注意：Python3.X 源码文件默认使用 utf-8 编码，所以可以正常解析中文，无需指定 UTF-8 编码。如果你使用编辑器，同时需要设置好编辑器的编码。

2. 在 python 里，标识符有字母、数字、下划线组成，所有标识符可以包括英文、数字以及下划线（_），但不能以数字开头。
3. python 中的标识符是区分大小写的。
4. 以下划线开头的标识符是有特殊意义的。

- 以单下划线开头的代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用"from xxx import *"而导入；
- 以双下划线开头的代表类的私有成员；
- 以双下划线开头和结尾的代表 python 里特殊方法专用的标识，如__init__（）代表类的构造函数。

5. python 最具特色的就是用缩进来写模块。缩进的空白数量是可变的，但是所有代码块语句必须包含相同的缩进空白数量，这个必须严格执行，要么全部是 tab 缩进，要么是全部空格缩进。
6. Python 语句中一般以新行作为为语句的结束符。但是我们可以使用斜杠（\）将一行的语句分为多行显示，如下所示：

```
total = item_one + \
        item_two + \
        item_three
```

语句中包含[], {} 或 () 括号就不需要使用多行连接符。如下实例：

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

7. Python 接收单引号(' ')，双引号(" ")，三引号(''' ''') 来表示字符串，引号的开始与结束必须的相同类型的。其中三引号可以由多行组成，编写多行文本的快捷语法，常用语文档字符串，在文件的特定地点，被当做注释。

```
word = 'word'
sentence = "这是一个句子。"
paragraph = """这是一个段落。
包含了多个语句"""
```

8. python 中单行注释采用 # 开头，多行注释使用三个单引号(''')或三个双引号(""")。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# 文件名：test.py


'''
这是多行注释，使用单引号。
这是多行注释，使用单引号。
这是多行注释，使用单引号。
'''

"""
这是多行注释，使用双引号。
这是多行注释，使用双引号。
这是多行注释，使用双引号。
"""
```

注意：多行注释和多行字符串语法相同，使用时区分就好。

9. 函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。类和函数入口之间也用一行空行分隔，以突出函数入口的开始。空行也是程序代码的一部分。

```
#coding=utf-8
#!/usr/bin/env python

class AnyIter(object):
    def __init__(self, data, safe=0):
        self.safe = safe
        self.iter = iter(data)

    def __iter__(self):
        return self
```

10. 下面的程序在按回车键后就会等待用户输入：

```
>>> a=raw_input("请输入名字：")
请输入名字：蓝张生
>>> print a
蓝张生
>>>
```

"\n\n"在结果输出前会输出两个新的空行。一旦用户按下键时，程序将退出。

11. Python 可以在同一行中使用多条语句，语句之间使用分号(;)分割，以下是一个简单的实例：

```
#!/usr/bin/python

import sys; x = 'runoob'; sys.stdout.write(x + '\n')
```

12. 缩进相同的一组语句构成一个代码块，我们称之代码组。像 if、while、def 和 class 这样的复合语句，首行以关键字开始，以冒号(:)结束，该行之后的一行或多行代码构成代码组。我们将首行及后面的代码组称为一个子句。如下实例：

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

13. 很多程序可以执行一些操作来查看一些基本信息，Python 可以使用-h 参数查看各参数帮助信息：

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit

[ etc. ]
```

第 2 章 变量类型

1. Python 中的变量赋值不需要类型声明：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```
counter = 100 # 赋值整型变量
miles = 1000.0 # 浮点型
name = "John" # 字符串

print counter
print miles
print name
```

2. Python 允许你同时为多个变量赋值。例如：

```
a = b = c = 1
```

也可以为多个对象指定多个变量：

```
a, b, c = 1, 2, "john"
```

3. Python 有五个标准的数据类型：

- Numbers（数字）
- String（字符串）
- List（列表）
- Tuple（元组）
- Dictionary（字典）

(1) Numbers（数字）：

当你指定一个值时，Number 对象就会被创建：

```
var1 = 1
var2 = 10
```

您也可以使用 del 语句删除一些对象的引用。del 语句的语法是：

```
del var1[,var2[,var3[....,varN]]]]
```

如

```
del var
del var_a, var_b
```

(2) String（字符串）

python 的字串列表有 2 种取值顺序：

- 从左到右索引默认 0 开始的，最大范围是字符串长度少 1。

- 从右到左索引默认-1 开始的，最大范围是字符串开头。

(3) List（列表）

列表可以完成大多数集合类的数据结构实现。它支持字符，数字，字符串甚至可以包含列表（所谓嵌套）。列表用[ ]标识。是 python 最通用的复合数据类型。看这段代码就明白。列表中的值得分割也可以用到变量[头下标:尾下标]，就可以截取相应的列表，从左到右索引默认 0 开始的，从右到左索引默认-1 开始，下标可以为空表示取到头或尾。加号（+）是列表连接运算符，星号（*）是重复操作。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list # 输出完整列表
print list[0] # 输出列表的第一个元素
print list[1:3] # 输出第二个至第三个的元素
print list[2:] # 输出从第三个开始至列表末尾的所有元素
print tinylist * 2 # 输出列表两次
print list + tinylist # 打印组合的列表
```

以上实例输出结果：

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

(4) Tuple（元组）

元组是另一个数据类型，类似于 List（列表）。元组用"()"标识。内部元素用逗号隔开。但是元组不能二次赋值，相当于只读列表。

(5) Dictionary（字典）

字典(dictionary)是除列表以外 python 之中最灵活的内置数据结构类型。列表是有序的对象结合，字典是无序的对象集合。两者之间的区别在于：字典当中的元素是通过键来存取的，而不是通过偏移存取。字典用"{ }"标识。字典由索引(key)和它对应的值 value 组成。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

dict = {}
```

```
dict['one'] = "This is one"
dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print dict['one'] # 输出键为'one' 的值
print dict[2] # 输出键为 2 的值
print tinydict # 输出完整的字典
print tinydict.keys() # 输出所有键
print tinydict.values() # 输出所有值
```

输出结果为：

```
This is one This is two {'dept': 'sales', 'code': 6734, 'name': 'john'} ['dept', 'code', 'name'] ['sales', 6734, 'john']
```

4. Python 支持切片语句。在 Python 中，加号（+）是字符串连接运算符，星号（*）是重复操作。如下实例：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

str = 'Hello World!'

print str # 输出完整字符串
print str[0] # 输出字符串中的第一个字符
print str[2:5] # 输出字符串中第三个至第五个之间的字符串
print str[2:] # 输出从第三个字符开始的字符串
print str * 2 # 输出字符串两次
print str + "TEST" # 输出连接的字符串
```

以上实例输出结果：

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

5. 下面几个函数可以执行数据类型之间的转换。这些函数返回一个新的对象，表示转换的值。

函数	描述
int(x [,base])	将 x 转换为一个整数
long(x [,base])	将 x 转换为一个长整数

float(x)	将 x 转换到一个浮点数
complex(real [,imag])	创建一个复数
str(x)	将对象 x 转换为字符串
repr(x)	用来计算在字符串中的有效 Python 表达式,并返回一个对象，完成和 str 相反的功能
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表
set(s)	转换为可变集合
dict(d)	创建一个字典。d 必须是一个序列 (key,value)元组。
frozenset(s)	转换为不可变集合
unichr(x)	将一个整数转换为 Unicode 字符
chr(x)	将一个整数转换为一个字符
ord(x)	将一个字符转换为它的整数值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串

第 3 章 运算符

1. python 支持的运算符和 C++类似，同时，还增加两种运算符：成员运算符和身份运算符：

- 成员运算符包括 in 和 not in：

运算符	描述	实例
in	如果在指定的序列中找到值返回 True，否则返回 False。	x 在 y 序列中 , 如果 x 在 y 序列中返回 True。
not in	如果在指定的序列中没有找到值返回 True，否则返回 False。	x 不在 y 序列中 , 如果 x 不在 y 序列中返回 True。

以下实例演示了 Python 所有成员运算符的操作：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

a = 10
b = 20
list = [1, 2, 3, 4, 5 ];

if ( a in list ):
```

```
  print "1 - 变量 a 在给定的列表中 list 中"
else:
  print "1 - 变量 a 不在给定的列表中 list 中"

if ( b not in list ):
  print "2 - 变量 b 不在给定的列表中 list 中"
else:
  print "2 - 变量 b 在给定的列表中 list 中"

# 修改变量 a 的值
a = 2
if ( a in list ):
  print "3 - 变量 a 在给定的列表中 list 中"
else:
  print "3 - 变量 a 不在给定的列表中 list 中"
```

以上实例输出结果：

```
1 - 变量 a 不在给定的列表中 list 中
2 - 变量 b 不在给定的列表中 list 中
3 - 变量 a 在给定的列表中 list 中
```

- 身份运算符用于比较两个对象的存储单元：

运算符	描述	实例
is	is 是判断两个标识符是不是引用自一个对象	x is y, 如果 id(x) 等于 id(y) , is 返回结果 1
is not	is not 是判断两个标识符是不是引用自不同对象	x is not y, 如果 id(x) 不等于 id(y). is not 返回结果 1

实例如下：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

a = 20
b = 20
c = 30

if ( a is b ):
  print "1 - a 和 b 有相同的标识"
else:
  print "1 - a 和 b 没有相同的标识"
if (a is c):
  print "1 - a 和 b 有相同的标识"
else:
  print "1 - a 和 b 没有相同的标识"
```

输出如下：

```
1 - a 和 b 有相同的标识
1 - a 和 b 没有相同的标识
```

2. pass 是空语句，是为了保持程序结构的完整性。pass 不做任何事情，一般用做占位语句。

第 4 章 数字

1. 可以使用 del 语句删除一些 Number 对象引用。

```
var1=10;
del var;
```

2. 随机数函数

函数	描述
choice(seq)	从序列的元素中随机挑选一个元素，比如 random.choice(range(10))，从 0 到 9 中随机挑选一个整数。
randrange ([start,] stop [,step])	从指定范围内，按指定基数递增的集合中获取一个随机数，基数缺省值为 1
random()	随机生成下一个实数，它在[0,1)范围内。
seed([x])	改变随机数生成器的种子 seed。如果你不了解其原理，你不必特别去设定 seed，Python 会帮你选择 seed。
shuffle(lst)	将序列的所有元素随机排序
uniform(x, y)	随机生成下一个实数，它在[x,y]范围内。

第 5 章 字符串

1. 创建字符串很简单，只要为变量分配一个值即可。例如：

```
var1 = 'Hello World!'
var2 = "Python Runoob"
```

2. 字符串需要使用引号来括住。

3. Python 不支持单字符类型，单字符也在 Python 也是作为一个字符串使用。Python 访问子字符串，可以使用方括号来截取字符串，如下实例：

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Runoob"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

以上实例执行结果：

```
var1[0]:  H
var2[1:5]:  ytho
```

4. 你可以对已存在的字符串进行修改，并赋值给另一个变量，如下实例：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

var1 = 'Hello World!'

print "更新字符串 :- ", var1[:6] + 'Runoob!'                    #截取--修改
```

以上实例执行结果

```
更新字符串 :-  Hello Runoob!
```

5. 字符串运算符（下表实例变量 a 值为字符串"Hello"，b 变量值为"Python"：）：

操作符	描述	实例
+	字符串连接	a + b 输出结果： HelloPython
*	重复输出字符串	a*2 输出结果：HelloHello
[]	通过索引获取字符串中字符	a[1] 输出结果 e
[:]	截取字符串中的一部分	a[1:4] 输出结果 ell
in	成员运算符 - 如果字符串中包含给定的字符返回 True	H in a 输出结果 1
not in	成员运算符 - 如果字符串中不包含给定的字符返回 True	M not in a 输出结果 1
r/R	原始字符串 - 原始字符串：所有的字符串都是直接按照字面的意思来使用，没有转义特殊或不能打印的字符。 原始字符串除在字符串的第一个引号前加上字母"r"（可以大小写）以外，与普通字	print r'\n' 输出 \n 和 print R'\n' 输出 \n

	符串有着几乎完全相同的语法。	
%	格式字符串	

6. 在 Python 中，字符串格式化使用与 C 中 sprintf 函数一样的语法。如下实例：

```
#!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

7. 字符串格式化符号：

符 号	描述
%c	格式化字符及其ASCII码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f和%e的简写
%G	%f 和 %E 的简写
%p	用十六进制数格式化变量的地址

格式化操作符辅助指令：

符号	功能
*	定义宽度或者小数点精度
-	用做左对齐
+	在正数前面显示加号(+)
<sp>	在正数前面显示空格
#	在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X'(取决于用的是'x'还是'X')
0	显示的数字前面填充'0'而不是默认的空格
%	'%%'输出一个单一的'%'
(var)	映射变量(字典参数)
m.n.	m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话)

8. 三引号可以用于定义多行字符串，这些字符串里可以包含换行符、制表符以及其他特殊字符。

```
 >>> hi = '''hi
there'''
>>> hi   # repr()
'hi\nthere'
>>> print hi  # str()
hi
there
```

三引号可以实现所见即所得的效果。一个典型的用例是，当你需要一块 HTML 或者 SQL 时，这时用字符串组合，特殊字符串转义将会非常的繁琐。

```
 errHTML = '''
<HTML><HEAD><TITLE>
Friends CGI Demo</TITLE></HEAD>
<BODY><H3>ERROR</H3>
<B>%s</B><P>
<FORM><INPUT TYPE=button VALUE=Back
ONCLICK="window.history.back()"></FORM>
</BODY></HTML>
'''
cursor.execute('''
CREATE TABLE users (
login VARCHAR(8),
uid INTEGER,
prid INTEGER)
''')
```

9. 三引号有两种：

- 3 个单引号。
- 3 个双引号。

10. Python 中定义一个 Unicode 字符串和定义一个普通字符串一样简单：

```
>>> u'Hello World !'
u'Hello World !'
```

引号前小写的"u"表示这里创建的是一个 Unicode 字符串。如果你想加入一个特殊字符，可以使用 Python 的 Unicode-Escape 编码。如下例所示：

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

被替换的 \u0020 标识表示在给定位置插入编码值为 0x0020 的 Unicode 字符。

第 6 章 列表

1. 序列是 Python 中最基本的数据结构。序列中的每个元素都分配一个数字 – 它的位置，或索引，第一个索引是 0，第二个索引是 1，依此类推。
2. 序列：成员有序排列的，并且可以通过下标偏移量访问到它的一个或者几个成员。序列包括下面这些： 字符串(普通字符串和 unicode 字符串)，列表，和元组类型。
3. 有七个序列类型：字符串、元组、列表、字节数组、缓冲区、Unicode 字符串和 xrange 对象。其中字符串和元组是不可变的序列类型：这类对象一旦创建不能修改。其它为可变序列类型。
4. 所有序列类型都适用的操作符：

```
x in s
x not in s
s + t
s * n, n * s
s[i]
s[i:j]
s[i:j:k]
len(s)
min(s)
max(s)
s.index(x)
s.count(x)
```

5. Python 有 6 个序列的内置类型，但最常见的是列表和元组。序列都可以进行的操作包括索引，切片，加，乘，检查成员。此外，Python 已经内置确定序列的长度以及确定最大和最小的元素的方法。
6. 列表的数据项不需要具有相同的类型。
7. 创建一个列表，只要把逗号分隔的不同的数据项使用方括号括起来即可。如下所示：

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"];
```

8. 使用下标索引来访问列表中的值，同样你也可以使用方括号的形式截取字符，如下所示：

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];

print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

以上实例输出结果：

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

9. 你可以对列表的数据项进行修改或更新，你也可以使用 append()方法来添加列表项，如下所示：

```
#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000];

print "Value available at index 2 : "
print list[2];
list[2] = 2001;
print "New value available at index 2 : "
print list[2];
```

以上实例输出结果：

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

10. 可以使用 del 语句来删除列表的的元素。

11. 列表对 + 和 * 的操作符与字符串相似。+ 号用于组合列表，* 号用于重复列表。

12. 函数和方法：

- 函数是一段代码，通过名字来进行调用。它能将一些数据（参数）传递进去进行处理，然后返回一些数据（返回值），也可以没有返回值。
- 方法也是一段代码，也通过名字来进行调用，但它跟一个对象相关联。方法在 C++中称为成员函数。

13. Python 列表函数

序号	函数
1	cmp(list1, list2) 比较两个列表的元素
2	len(list) 列表元素个数
3	max(list) 返回列表元素最大值
4	min(list) 返回列表元素最小值
5	list(seq) 将元组转换为列表

14. Python 包含以下方法：

序号	方法
1	list.append(obj) 在列表末尾添加新的对象
2	list.count(obj) 统计某个元素在列表中出现的次数
3	list.extend(seq) 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
4	list.index(obj) 从列表中找出某个值第一个匹配项的索引位置
5	list.insert(index, obj) 将对象插入列表
6	list.pop(obj=list[-1]) 移除列表中的一个元素（默认最后一个元素），并且返回该元素的值
7	list.remove(obj) 移除列表中某个值的第一个匹配项
8	list.reverse() 反向列表中元素
9	list.sort([func]) 对原列表进行排序

注：list.sort()中的 list 其实就是列表。

第 7 章 元组

1. Python 的元组与列表类似，不同之处在于元组的元素可以修改。元组使用小括号，列表使用方括号。元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

元组中只包含一个元素时，需要在元素后面添加逗号：

```
tup1 = (50,);
```

2. 元组可以使用下标索引来访问元组中的值。元组中的元素值是不允许修改的，但我们可以对元组进行连接组合。
3. 元组中的元素值是不允许删除的，但我们可以使用 del 语句来删除整个元组。
4. 任意无符号的对象，以逗号隔开，默认为元组，如下实例：

```
#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz';
```

```
x, y = 1, 2;
print "Value of x , y : ", x,y;
```

以上实例运行结果：

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

5. 元组包含了以下内置函数

序号	方法及描述
1	cmp(tuple1, tuple2) 比较两个元组元素。
2	len(tuple) 计算元组元素个数。
3	max(tuple) 返回元组中元素最大值。
4	min(tuple) 返回元组中元素最小值。
5	tuple(seq) 将列表转换为元组。

第 8 章 字典

1. 字典是另一种可变容器模型，且可存储任意类型对象。字典的每个键值(key=>value)对用冒号(:)分割，每个对之间用逗号(,)分割，整个字典包括在花括号({})中 ,格式如下所示：

```
d = {key1 : value1, key2 : value2 }
```

键必须是唯一的，但值则不必。值可以取任何数据类型，但键必须是不可变的，如字符串，数字或元组。创建字典：

```
dict1 = { 'abc': 456 };
dict2 = { 'abc': 123, 98.6: 37 };
```

2. 访问字典里的值，需要把相应的键放入熟悉的方括弧：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Name']: ", dict['Name'];
print "dict['Age']: ", dict['Age'];
```

3. 向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对如下实例：

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School";                                    # Add new entry
```

4. 能删单一的元素也能清空字典，清空只需一项操作。删除一个字典用 del 命令。

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

del dict['Name']; # 删除键是'Name'的条目
dict.clear();     # 清空词典所有条目
del dict ;        # 删除词典
```

5. 字典值可以没有限制地取任何 python 对象，既可以是标准的对象，也可以是用户定义的，但键不行。两个重要的点需要记住：

- 不允许同一个键出现两次，字典中已经存在的键值会被覆盖掉。
- 键必须不可变，所以可以用数字，字符串或元组充当，所以用列表就不行。

6. 字典方法

序号	函数及描述
1	radiansdict.clear() 删除字典内所有元素
2	radiansdict.copy() 返回一个字典的浅复制
3	radiansdict.fromkeys() 创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值
4	radiansdict.get(key, default=None) 返回指定键的值，如果值不在字典中返回default值
5	radiansdict.has_key(key) 如果键在字典dict里返回true，否则返回false
6	radiansdict.items() 以列表返回可遍历的(键, 值) 元组数组
7	radiansdict.keys() 以列表返回一个字典所有的键
8	radiansdict.setdefault(key, default=None) 和get()类似, 但如果键不存在于字典中，将会添加键并将值设为default
9	radiansdict.update(dict2) 把字典dict2的键/值对更新到dict里
10	radiansdict.values() 以列表返回字典中的所有值

注：　　radiansdict 是对象名。

7. 字典函数：

序号	函数及描述
1	cmp(dict1, dict2) 比较两个字典元素。
2	len(dict) 计算字典元素个数，即键的总数。
3	str(dict) 输出字典可打印的字符串表示。
4	type(variable) 返回输入的变量类型，如果变量是字典就返回字典类型。

第 9 章 日期和时间

1. Python 提供了一个 time 和 calendar 模块可以用于格式化日期和时间。时间间隔是以秒为单位的浮点小数。

2. time 模块下有很多函数可以转换常见日期格式。如函数 time.time()用于获取当前时间戳：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import time;  # 引入 time 模块

ticks = time.time()
print "当前时间戳为:", ticks
```

以上实例输出结果：

```
当前时间戳为: 1459994552.51
```

3. 很多 Python 函数用 struct_time 装 9 组数字处理时间，这种结构具有如下属性：

序号	属性	值
0	tm_year	2008
1	tm_mon	1 到 12
2	tm_mday	1 到 31
3	tm_hour	0 到 23
4	tm_min	0 到 59
5	tm_sec	0 到 61 (60或61 是闰秒)
6	tm_wday	0到6 (0是周一)
7	tm_yday	1 到 366(儒略历)
8	tm_isdst	-1, 0, 1, -1是决定是否为夏令时的旗帜

4. localtime()用于将时间转换为本地时间。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import time

localtime = time.localtime(time.time())
print "本地时间为 :", localtime
```

以上实例输出结果：

```
本地时间为 : time.struct_time(tm_year=2016, tm_mon=4, tm_mday=7, tm_hour=10, tm_min=3, tm_sec=27, tm_wday=3, tm_yday=98, tm_isdst=0)
```

5. asctime()函数接受时间元组，并返回一个可读的时间形式，类似于"Tue Dec 11 18:07:14 2008"：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import time

localtime = time.asctime( time.localtime(time.time()) )
```

```
print "本地时间为 :", localtime
```

以上实例输出结果：

```
本地时间为 : Thu Apr  7 10:05:21 2016
```

先用 time.time()生成当前的时间戳，然后使用 time.localtime()转化本地时间，再用 time.asctime()生成可读的时间模式。

6. time 模块的 strftime 方法用于格式化日期：

```
time.strftime(format[, t])
```

实例如下：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import time

# 格式化成 2016-03-20 11:45:39 形式
print time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())

# 格式化成 Sat Mar 28 22:24:24 2016 形式
print time.strftime("%a %b %d %H:%M:%S %Y", time.localtime())

# 将格式字符串转换为时间戳
a = "Sat Mar 28 22:24:24 2016"
print time.mktime(time.strptime(a,"%a %b %d %H:%M:%S %Y"))
```

以上实例输出结果：

```
2016-04-07 10:25:09
Thu Apr 07 10:25:09 2016
1459175064.0
```

7. Calendar 模块用于处理年历和月历，例如打印某月的月历：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import calendar

cal = calendar.month(2016, 1)
print "以下输出 2016 年 1 月份的日历:"
print cal;
```

以上实例输出结果：

```
以下输出 2016 年 1 月份的日历:
    January 2016
Mo Tu We Th Fr Sa Su
```

```
      1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

第 10 章 函数

1. 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。

```
def functionname( parameters ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

2. 所有参数（自变量）在 Python 里都是按引用传递。如果你在函数里修改了参数，那么在调用这个函数的函数里，原始的参数也被改变了。
3. 以下是调用函数时可使用的正式参数类型：

- 必备参数
- 关键字参数：关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

#可写函数说明
def printme( str ):
   "打印任何传入的字符串"
   print str;
   return;

#调用 printme 函数
printme( str = "My string");                    #给参数赋值
```

以上实例输出结果：

```
My string
```

注意：在调用时给 str 赋值是关键字参数，在定义时赋值是默认参数。

- 默认参数

- 不定长参数：不定长参数，和上述 2 种参数不同，声明时不会命名。基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

加了星号（*）的变量名会存放所有未命名的变量参数。选择不多传参数也可。

4. python 使用 lambda 来创建匿名函数。

- lambda 只是一个表达式，函数体比 def 简单很多。
- lambda 的主体是一个表达式，而不是一个代码块。仅仅能在 lambda 表达式中封装有限的逻辑进去。
- lambda 函数拥有自己的命名空间，且不能访问自有参数列表之外或全局命名空间里的参数。
- 虽然 lambda 函数看起来只能写一行，却不等同于 C 或 C++的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

5. lambda 函数的语法只包含一个语句，如下：

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

如下实例：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 可写函数说明
sum = lambda arg1, arg2: arg1 + arg2;

# 调用 sum 函数
print "相加后的值为 : ", sum( 10, 20 )                    #注意这种调用方式
print "相加后的值为 : ", sum( 20, 20 )
```

以上实例输出结果：

```
相加后的值为 :  30
相加后的值为 :  40
```

第 11 章 模块

1. 模块也是 Python 对象。
2. 想使用 Python 源文件，只需在另一个源文件里执行 import 语句，语法如下：

```
import module1[, module2[,... moduleN]
```

3. 一个模块只会被导入一次，不管你执行了多少次 import。这样可以防止导入模块被一遍又一遍地执行。

4. Python 的 from 语句让你从模块中导入一个指定的部分到当前命名空间中。语法如下：

```
from modname import name1[, name2[, ... nameN]]
```

例如，要导入模块 fib 的 fibonacci 函数，使用如下语句：

```
from fib import fibonacci
```

5. 使用*号把一个模块的所有内容全都导入到当前的命名空间也：

```
from modname import *
```

6. dir()函数返回一个排好序的字符串列表，内容是一个模块里定义过的名字。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 导入内置 math 模块
import math

content = dir(math)

print content;
```

以上实例输出结果：

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

7. 当一个模块被导入到一个脚本，模块顶层部分的代码只会被执行一次。因此，如果你想重新执行模块里顶层部分的代码，可以用 reload()函数。该函数会重新导入之前导入过的模块。语法如下：

```
reload(module_name)
```

第 12 章 文件 i/o

1. 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。
2. Python 提供了两个内置函数从标准输入读入一行文本，默认的标准输入是键盘。如下：

- raw_input：raw_input([prompt]) 函数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。
- input：input([prompt]) 函数和 raw_input([prompt]) 函数基本类似，但是 input 可以接收一个 Python 表达式作为输入，并将运算结果返回。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

str = input("请输入：");
print "你输入的内容是: ", str
```

这会产生如下的对应着输入的结果：

```
请输入：[x*5 for x in range(2,10,2)]
你输入的内容是:  [10, 20, 30, 40]
```

3. Python 提供了必要的函数和方法进行默认情况下的文件基本操作。你可以用 file 对象做大部分的文件操作。
4. 你必须先用 Python 内置的 open()函数打开一个文件，创建一个 file 对象，相关的方法才可以调用它进行读写。语法：

```
file object = open(file_name [, access_mode][, buffering])
```

各个参数的细节如下：

- file_name：file_name 变量是一个包含了你要访问的文件名称的字符串值。
- access_mode：access_mode 决定了打开文件的模式：只读，写入，追加等。所有可取值见如下的完全列表。这个参数是非强制的，默认文件访问模式为只读(r)。
- buffering:如果 buffering 的值被设为 0，就不会有寄存。如果 buffering 的值取 1，访问文件时会寄存行。如果将 buffering 的值设为大于 1 的整数，表明了这就是的寄存区的缓冲大小。如果取负值，寄存区的缓冲大小则为系统默认。

5. 一个文件被打开后，你有一个 file 对象，你可以得到有关该文件的各种信息。以下是和 file 对象相关的所有属性的列表：

属性	描述
file.closed	返回true如果文件已被关闭，否则返回false。
file.mode	返回被打开文件的访问模式。
file.name	返回文件的名称。
file.softspace	如果用print输出后，必须跟一个空格符，则返回false。否则返回true。

6. File 对象的 close()方法刷新缓冲区里任何还没写入的信息，并关闭该文件，这之后便不能再进行写入。当一个文件对象的引用被重新指定给另一个文件时，Python 会关闭之前的文件。用 close()方法关闭文件是一个很好的习惯。语法：

```
fileObject.close();
```

7. write()方法可将任何字符串写入一个打开的文件。需要重点注意的是，Python 字符串可以是二进制数据，而不是仅仅是文字。write()方法不会在字符串的结尾添加换行符('\n')。语法：

```
fileObject.write(string);
```

8. read()方法从一个打开的文件中读取一个字符串。需要重点注意的是，Python 字符串可以是二进制数据，而不是仅仅是文字。语法：

```
fileObject.read([count]);
```

9. tell()方法告诉你文件内的当前位置；换句话说，下一次的读写会发生在文件开头这么多字节之后。seek（offset [,from]）方法改变当前文件的位置。Offset 变量表示要移动的字节数。From 变量指定开始移动字节的参考位置。
10. Python 的 os 模块提供了帮你执行文件处理操作的方法，比如重命名和删除文件。要使用这个模块，你必须先导入它，然后才可以调用相关的各种功能。rename()方法：rename()方法需要两个参数，当前的文件名和新文件名。语法：

```
os.rename(current_file_name, new_file_name)
```

11. 你可以用 remove()方法删除文件，需要提供要删除的文件名作为参数。语法：

```
os.remove(file_name)
```

12. 可以使用 os 模块的 mkdir()方法在当前目录下创建新的目录们。你需要提供一个包含了要创建的目录名称的参数。语法：

```
os.mkdir("newdir")
```

13. 可以用 chdir()方法来改变当前的目录。chdir()方法需要的一个参数是你想设成当前目录的目录名称。语法：

```
os.chdir("newdir")
```

14. getcwd()方法显示当前的工作目录。语法：

```
os.getcwd()
```

15. rmdir()方法删除目录，目录名称以参数传递。在删除这个目录之前，它的所有内容应该先被清除。语法：

```
os.rmdir('dirname')
```

第 13 章 File(文件) 方法

1. file.close()：关闭文件。关闭后文件不能再进行读写操作。
2. file.flush()：刷新文件内部缓冲，直接把内部缓冲区的数据立刻写入文件, 而不是被动的等待输出缓冲区写入。
3. file.fileno()：返回一个整型的文件描述符(file descriptor FD 整型), 可以用在如 os 模块的 read 方法等一些底层操作上。
4. file.isatty()：如果文件连接到一个终端设备返回 True，否则返回 False。
5. file.next()：返回文件下一行。
6. file.read([size])：从文件读取指定的字节数，如果未给定或为负则读取所有。
7. file.readline([size])：读取整行，包括 "\n" 字符。
8. file.readlines([sizehint])：读取所有行并返回列表，若给定 sizeint>0，返回总和大约为 sizeint 字节的行, 实际读取值可能比 sizhint 较大, 因为需要填充缓冲区。
9. file.seek(offset[, whence])：设置文件当前位置
10. file.tell()：返回文件当前位置。
11. file.truncate([size])：截取文件，截取的字节通过 size 指定，默认为当前文件位置。

12. file.write(str)：将字符串写入文件，没有返回值。

13. file.writelines(sequence)：向文件写入一个序列字符串列表，如果需要换行则要自己加入每行的换行符。

第 14 章 异常处理

1. python 提供了两个非常重要的功能来处理 python 程序在运行中出现的异常和错误。你可以使用该功能来调试 python 程序。

- 异常处理:
- 断言(Assertions)

2. 捕捉异常可以使用 try/except 语句。try/except 语句用来检测 try 语句块中的错误，从而让 except 语句捕获异常信息并处理。如果你不想在异常发生时结束你的程序，只需在 try 里捕获它。以下为简单的 try....except...else 的语法：

```
try:
   <语句>       #运行别的代码
except <名字>：
   <语句>       #如果在 try 部份引发了'name'异常
except <名字>，<数据>:
   <语句>       #如果引发了'name'异常，获得附加的数据
else:
   <语句>       #如果没有异常发生
```

try 的工作原理是，当开始一个 try 语句后，python 就在当前程序的上下文中作标记，这样当异常出现时就可以回到这里，try 子句先执行，接下来会发生什么依赖于执行时是否出现异常。

- 如果当 try 后的语句执行时发生异常，python 就跳回到 try 并执行第一个匹配该异常的 except 子句，异常处理完毕，控制流就通过整个 try 语句（除非在处理异常时又引发新的异常）。
- 如果在 try 后的语句里发生了异常，却没有匹配的 except 子句，异常将被递交到上层的 try，或者到程序的最上层（这样将结束程序，并打印缺省的出错信息）。
- 如果在 try 子句执行时没有发生异常，python 将执行 else 语句后的语句（如果有 else 的话），然后控制流通过整个 try 语句。

下面是简单的例子，它打开一个文件，在该文件中的内容写入内容，但文件没有写入权限，发生了异常：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```
try:
    fh = open("testfile", "w")
    fh.write("这是一个测试文件，用于测试异常!!")
except IOError:
    print "Error: 没有找到文件或读取文件失败"
else:
    print "内容写入文件成功"
    fh.close()
```

执行结果如下：

```
$ python test.py
Error: 没有找到文件或读取文件失败
```

3. 你可以不带任何异常类型使用 except，如下实例：

```
try:
    正常的操作
   ......................
except:
    发生异常，执行这块代码
   ......................
else:
    如果没有异常执行这块代码
```

以上方式 try-except 语句捕获所有发生的异常。但这不是一个很好的方式，我们不能通过该程序识别出具体的异常信息。因为它捕获所有的异常。

4. 你也可以使用相同的 except 语句来处理多个异常信息，如下所示：

```
try:
    正常的操作
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   发生以上多个异常中的一个，执行这块代码
   ......................
else:
    如果没有异常执行这块代码
```

5. try-finally 语句无论是否发生异常都将执行最后的代码。

```
try:
    <语句>
finally:
    <语句>    #退出 try 时总会执行
```

```
raise
```

6. 一个异常可以带上参数，可作为输出的异常信息参数。你可以通过 except 语句来捕获异常的参数，如下所示：

```
try:
   正常的操作
  ......................
except ExceptionType, Argument:
   你可以在这输出 Argument 的值...
```

7. 我们可以使用 raise 语句自己触发异常。raise 语法格式如下：

```
raise [Exception [, args [, traceback]]]
```

语句中 Exception 是异常的类型（例如，NameError）参数是一个异常参数值。该参数是可选的，如果不提供，异常的参数是"None"。最后一个参数是可选的（在实践中很少使用），如果存在，是跟踪异常对象。

8. 通过创建一个新的异常类，程序可以命名它们自己的异常。异常应该是典型的继承自 Exception 类，通过直接或间接的方式。以下为与 RuntimeError 相关的实例,实例中创建了一个类，基类为 RuntimeError，用于在异常触发时输出更多的信息。在 try 语句块中，用户自定义的异常后执行 except 块语句，变量 e 是用于创建 Networkerror 类的实例。

```
class Networkerror(RuntimeError):
   def __init__(self, arg):
      self.args = arg
```

在你定义以上类后，你可以触发该异常，如下所示：

```
try:
   raise Networkerror("Bad hostname")
except Networkerror,e:
   print e.args
```

第 15 章 面向对象

1. 方法重写：如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重写。

   实例变量：定义在方法中的变量，只作用于当前实例的类。

2. 使用 class 语句来创建一个新类，class 之后为类的名称并以冒号结尾，如下实例：

```
class ClassName:
   '类的帮助信息'   #类文档字符串
   class_suite  #类体
```

类的帮助信息可以通过 ClassName.__doc__查看。class_suite 由类成员，方法，数据属性组成。

以下是一个简单的 Python 类实例：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

class Employee:
   '所有员工的基类'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- empCount 变量是一个类变量，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用 Employee.empCount 访问。
- 第一种方法__init__()方法是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法。

要创建一个类的实例，你可以使用类的名称，并通过__init__方法接受参数：

```
"创建 Employee 类的第一个对象"
emp1 = Employee("Zara", 2000)
"创建 Employee 类的第二个对象"
emp2 = Employee("Manni", 5000)
```

3. 您可以使用点(.)来访问对象的属性。使用如下类的名称访问类变量：
4. 可以添加，删除，修改类的属性，如下所示：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```
class Employee:
      '所有员工的基类'
       empCount = 0

def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

def displayCount(self):
    print "Total Employee %d" % Employee.empCount

def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"创建 Employee 类的第一个对象"
emp1 = Employee("Zara", 2000)
emp1.age = 7                          # 添加一个 'age' 属性
emp1.age = 8                          # 修改 'age' 属性
```

5. 可以使用以下函数的方式来访问属性：

- getattr(obj, name[, default]) : 访问对象的属性。
- hasattr(obj,name) : 检查是否存在一个属性。
- setattr(obj,name,value) : 设置一个属性。如果属性不存在，会创建一个新属性。
- delattr(obj, name) : 删除属性。

6. Python 内置类属性

- __dict__ : 类的属性
- __doc__ :类的文档字符串
- __name__: 类名
- __module__: 类定义所在的模块（类的全名是'__main__.className'，如果类位于一个导入模块 mymod 中，那么 className.__module__ 等于 mymod）
- __bases__ : 类的所有父类构成元素（包含了一个由所有父类组成的元组）

7. 同 Java 语言一样，Python 使用了引用计数这一简单技术来追踪内存中的对象。在 Python 内部记录着所有使用中的对象各有多少引用。一个内部跟踪变量，称为一个引用计数器。当对象被创建时， 就创建了一个引用计数， 当这个对象不再需要时， 也

就是说， 这个对象的引用计数变为 0 时， 它被垃圾回收。但是回收不是"立即"的，由解释器在适当的时机，将垃圾对象占用的内存空间回收。

```
a = 40      # 创建对象  <40>
b = a       # 增加引用， <40> 的计数
c = [b]     # 增加引用. <40> 的计数

del a       # 减少引用 <40> 的计数
b = 100     # 减少引用 <40> 的计数
c[0] = -1   # 减少引用 <40> 的计数
```

8. 析构函数 __del__ ， __del__在对象销毁的时候被调用，当对象不再被使用时，__del__方法运行。
9. 如果在继承元组中列了一个以上的类，那么它就被称作"多重继承" ：

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
    'Optional class documentation string'
    class_suite
```

10. 在 python 中继承中的一些特点：

- 在继承中基类的构造（__init__()方法）不会被自动调用，它需要在其派生类的构造中亲自专门调用。
- 在调用基类的方法时，需要加上基类的类名前缀，且需要带上 self 参数变量。区别于在类中调用普通函数时并不需要带上 self 参数
- Python 总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找。先在本类中查找调用的方法，找不到才去基类中找。

11. 如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

class Parent:        # 定义父类
    def myMethod(self):
        print '调用父类方法'

class Child(Parent): # 定义子类
    def myMethod(self):
        print '调用子类方法'
```

```
c = Child()         # 子类实例
c.myMethod()        # 子类调用重写方法
```

12. 下表列出了一些通用的功能，你可以在自己的类重写：

序号	方法, 描述 & 简单的调用
1	**__init__ (self [,args...])** 构造函数 简单的调用方法: *obj = className(args)*
2	**__del__(self)** 析构方法, 删除一个对象 简单的调用方法 : *dell obj*
3	**__repr__(self)** 转化为供解释器读取的形式 简单的调用方法 : *repr(obj)*
4	**__str__(self)** 用于将值转化为适于人阅读的形式 简单的调用方法 : *str(obj)*
5	**__cmp__ (self, x)** 对象比较 简单的调用方法 : *cmp(obj, x)*

13. __private_attrs：两个下划线开头，声明该属性为私有，不能在类地外部被使用或直接访问。在类内部的方法中使用时 self.__private_attrs。
14. 在类地内部，使用 def 关键字可以为类定义一个方法，与一般函数定义不同，类方法必须包含参数 self，且为第一个参数。
15. 不同于 java，python 构造子类对象时并不一定会自动调用父类的构造方法：

- 如果子类没有重写了父类的__init__()方法，则构造子类对象时 Python 会自动调用父的__init__()方法，需要传入父类__init__()方法的参数，否则会报错。
- 如果子类重写了父类的__init__()方法，则子类不会自动调用父类的__init__()方法，如果需要调用父类的__init__()方法，则可以使用 super 关键字来实现。

```
class Parent(object):
   def __init__(self, name):
      self.name = name
      print("create an instance of:", self.__class__.__name__)
      print("name attribute is:", self.name)
#子类继承父类
class Child(Parent):
```

```
    #子类中没有显示调用父类的初始化函数
    def __init__(self):
        print("call __init__ from Child class")
#c = Child("init Child")
#print()
#将子类实例化
c = Child()
print(c.name)
```

16.使用 super 关键字初始化父类，后接父类的方法名：

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
```

Student 类继承自 Person 类：

```
class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score
```

函数 super(Student, self)将返回当前类继承的父类，即 Person ，然后调用 __init__()方法，注意 self 参数已在 super()中传入，在__init__()中将隐式传递，不需要写出。super 是一个类，这个类要求最顶层的父类一定要继承于 object。

17.类属性和实例属性不同，类属性和实例无关。如果在类上绑定一个属性，则所有实例都可以访问类的属性，并且，所有实例访问的类属性都是同一个！也就是说，实例属性每个实例各自拥有，互相独立，而类属性有且只有一份。

```
class Person(object):
    address = 'Earth'                    #类属性
    def __init__(self, name):
        self.name = name
```

因为类属性是直接绑定在类上的，所以，访问类属性不需要创建实例，就可以直接访问：

```
print Person.address
# => Earth
```

如果定义了一个 Person 实例，那么在实例中 address 就是实例属性，而非类属性。所以判断是否是类属性的关键是调用者是类还是对象。当实例属性和类属性重名时，实例属性优先级高，它将屏蔽掉对类属性的访问。

18.通过@classmethod 定义类方法：

```
class Person(object):
    count = 0
    @classmethod
    def how_many(cls):
        return cls.count
    def __init__(self, name):
        self.name = name
        Person.count = Person.count + 1

print Person.how_many()
p1 = Person('Bob')
print Person.how_many()
```

19.

第 16 章 正则表达式

1. re 模块使 Python 语言拥有全部的正则表达式功能。
2. re.match 尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功的话，match()就返回 none。函数语法：

```
re.match(pattern, string, flags=0)
```

函数参数说明：

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。

3. re.search 扫描整个字符串并返回第一个成功的匹配。
4. re.match 只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回 None；而 re.search 匹配整个字符串，直到找到一个匹配。
5. Python 的 re 模块提供了 re.sub 用于替换字符串中的匹配项。

第 17 章 MySQL

1. Python 标准数据库接口为 Python DB-API，Python DB-API 为开发人员提供了数据库应用编程接口。不同的数据库你需要下载不同的 DB API 模块，例如你需要访问 Oracle 数据库和 Mysql 数据，你需要下载 Oracle 和 MySQL 数据库模块。
2. DB-API 是一个规范. 它定义了一系列必须的对象和数据库存取方式, 以便为各种各样的底层数据库系统和多种多样的数据库接口程序提供一致的访问接口 。Python 的 DB-API，为大多数的数据库实现了接口，使用它连接各数据库后，就可以用相同的方式操作各数据库。
3. Python DB-API 使用流程：

(1) 引入 API 模块。

(2) 获取与数据库的连接。

(3) 执行 SQL 语句和存储过程。

(4) 关闭数据库连接。

4. 为了用 DB-API 编写 MySQL 脚本，必须确保已经安装了 MySQL。复制以下代码，并执行：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import MySQLdb
```

第 18 章 网络编程

1. Python 提供了两个级别访问的网络服务：

- 低级别的网络服务支持基本的 Socket，它提供了标准的 BSD Sockets API，可以访问底层操作系统 Socket 接口的全部方法。
- 高级别的网络服务模块 SocketServer， 它提供了服务器中心类，可以简化网络服务器的开发。

2. Python 中，我们用 socket（）函数来创建套接字，语法格式如下：

```
socket.socket([family[, type[, proto]]])
```

第 19 章 多线程

1. 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。每个线程都有他自己的一组 CPU 寄存器，称为线程的上下文，该上下文反映了线程上次运行该线程的 CPU 寄存器的状态。
2. Python 中使用线程有两种方式：函数或者用类来包装线程对象。
3. 函数式：调用 thread 模块中的 start_new_thread()函数来产生新线程。语法如下:

```
thread.start_new_thread ( function, args[, kwargs] )
```

4. 线程的结束一般依靠线程函数的自然结束；也可以在线程函数中调用 thread.exit()，他抛出 SystemExit exception，达到退出线程的目的。
5. Python 通过两个标准库 thread 和 threading 提供对线程的支持。thread 提供了低级别的、原始的线程以及一个简单的锁。除了使用方法外，线程模块同样提供了 Thread 类来处理线程。
6. 使用 Threading 模块创建线程，直接从 threading.Thread 继承，然后重写__init__方法和 run 方法。

第 20 章 Python2.x 与 3x 版本区别

1. print 语句没有了，取而代之的是 print()函数。 Python 2.6 与 Python 2.7 部分地支持这种形式的 print 语法。在 Python 2.6 与 Python 2.7 里面，以下三种形式是等价的：

```
print "fish"
print ("fish") #注意 print 后面有个空格
print("fish") #print()不能带有任何其它参数
```

2. Python 2 没有字节（byte）型，在 Python 3，我们最终有了 Unicode (utf-8) 字符串，以及一个字节类：byte 和 bytearrays。由于 Python3.X 源码文件默认使用 utf-8 编码，这就使得以下代码是合法的：

```
>>> 中国 = 'china'
>>>print(中国)
china
```

3. 在 Python 3 中处理异常也轻微的改变了，在 Python 3 中我们现在使用 as 作为关键词。捕获异常的语法由 except exc, var 改为 except exc as var。使用语法 except (exc1, exc2) as var 可以同时捕获多种类别的异常。 Python 2.6 已经支持这两种语法。

- 在 2.x 时代，所有类型的对象都是可以被直接抛出的，在 3.x 时代，只有继承自 BaseException 的对象才可以被抛出。
- 2.x raise 语句使用逗号将抛出对象类型和参数分开，3.x 取消了这种奇葩的写法，直接调用构造函数抛出对象即可。

4. 八进制数必须写成 0o777，原来的形式 0777 不能用了；二进制必须写成 0b111。新增了一个 bin()函数用于将一个整数转换成二进制字串。 Python 2.6 已经支持这两种语法。在 Python 3.x 中，表示八进制字面量的方式只有一种，就是 0o1000。
5. Python 2.x 中不等于有两种写法 != 和 <>，Python 3.x 中去掉了<>, 只有!=一种写法。
6. Python 2.x 中反引号``相当于 repr 函数的作用。Python 3.x 中去掉了``这种写法，只允许使用 repr 函数
7. Py3.X 去除了 long 类型，现在只有一种整型 int，但它的行为就像 2.X 版本的 long。

Python 核心编程

第 1 章~第 10 章 python 基础

1. 乘方运算符为双星号(**)。python 提供了一个用于实现整除的操作符——双斜线。
2. Python 支持使用成对的单引号或双引号，三引号（三个连续的单引号或者双引号）可以用来包含特殊字符。
3. 使用索引运算符([ ])和切片运算符([:])可以得到子字符串。字符串有其特有的索引规则：第一个字符的索引是 0，最后一个字符的索引是-1。加号(+)用于字符串连接运算，星号(*)则用于字符串重复。

```
>>> pystr = 'Python'
>>> iscool = 'is cool!'
>>> pystr[0]
'P'
>>> pystr[2:5]                    #pystr = 'Python'
'tho'
>>> iscool[:2]
```

```
'is'
>>> iscool[3:]
'cool!'
>>> iscool[-1]
'!'
>>> pystr + iscool
'Pythonis cool!'
>>> pystr + ' ' + iscool
'Python is cool!'
>>> pystr * 2
'PythonPython'
>>> '-' * 20
'--------------------'
>>> pystr = '''python
... is cool'''
>>> pystr
'python\nis cool'
>>> print pystr
python
is cool
>>>
```

4. 字典是 Python 中的映射数据类型，由键-值(key-value)对构成。几乎所有类型的 Python 对象都可以用作键，不过一般还是以数字或者字符串最为常用。值可以是任意类型的 Python 对象，字典元素用大括号({ })包裹。

```
>>> aDict = {'host': 'earth'} # create dict
>>> aDict['port'] = 80 # add to dict
>>> aDict
{'host': 'earth', 'port': 80}
>>> aDict.keys()
['host', 'port']
>>> aDict['host']
'earth'
>>> for key in aDict:
... print key, aDict[key]
...
host earth
port 80
```

5. print 语句默认会给每一行添加一个换行符。只要在 print 语句的最后添加一个逗号(，)， 就可以改变它这种行为，带逗号的 print 语句输出的元素之间会自动添加一个空格。

```
print 'I like to use the Internet for:'
for item in ['e-mail', 'net-surfing', 'homework', 'chat']:
print item,
print
```

6. 函数的参数可以有一个默认值， 如果提供有默认值，在函数定义中， 参数以赋值语句的形式提供。事实上这仅仅是提供默认参数的语法， 它表示函数调用时如果没有提供这个参数， 它就取这个值做为默认值。

```
>>> def foo(debug=True):
```

7. 类是面向对象编程的核心， 它扮演相关数据及逻辑的容器角色。它们提供了创建“真实”对象(也就是实例)的蓝图。 Python 并不强求你以面向对象的方式编程。如何定义类：

```
class ClassName(base_class[es]):
    "optional documentation string"
    static_member_declarations
    method_declarations
```

使用 class 关键字定义类。 可以提供一个可选的父类或者说基类; 如果没有合适的基类，那就使用 object 作为基类。class 行之后是可选的文档字符串， 静态成员定义， 及方法定义。如何创建类实例：

```
class FooClass(object):
……

>>> foo1 = FooClass()
Created a class instance for John Doe
```

创建一个类实例就像调用一个函数， 它们确实拥有一样的语法。既然我们成功创建了第一个类实例， 那现在来进行一些方法调用：

```
>>> foo1.showname() Your name is John Doe
My name is __main__.FooClass
>>>
```

8. Python 用下划线作为变量前缀和后缀指定特殊变量。下划线的特殊用法总结：

- _xxx 不用'from module import *'导入
- __xxx__系统定义名字
- __xxx 类中的私有变量名

核心风格：避免用下划线作为变量名的开始，因为下划线对解释器有特殊的意义，而且是内建标识符所使用的符号，我们建议程序员避免用下划线作为变量名的开始。

10. 如果模块是被导入， __name__ 系统变量的值为模块名字。如果模块是被直接执行，__name__ 的值为 '__main__'。

11. 切片语法：

```
>>> foostr = 'abcde'
>>> foostr[::-1]
'edcba'
```

第三个元素是步长，步长为 2 表示从开始每隔 1 个元素取值。步长不能为 0，但可以为负数，表示从右到左取值。

12. 布尔逻辑运算符 and， or 和 not 都是 Python 关键字。not 运算符拥有最高优先级，只比所有比较运算符低一级。 and 和 or 运算符则相应的再低一级。

```
>>> not (x is y)
True
```

13. 工厂函数看上去有点像函数，实质上返回的是类，当你调用它们时，实际上是生成了该类型的一个实例，就像工厂生产货物一样。

```
class JSONConnector:
  def __init__(self, filepath):

class XMLConnector:
  def __init__(self, filepath):

def connector_factory(filepath):
  if filepath.endswith('json'):
    connector = JSONConnector
  elif filepath.endswith('xml'):
    connector = XMLConnector
  else:
    raise ValueError('Cannot connect to {}'.format(filepath))
```

```
return connector(filepath)
```

connector_factory 是一个工厂方法。

14. 如果你实际上想删除一个数值对象的引用， 使用 del 语句。

15. 字符串 A + 字符串 B 并不表示加法操作， 它表示的是把这两个字符串连接起来， 生成一个新的字符串。列表和字符串无法连接在一起，尽管都是序列，也就是说，两种相同类型有序列才能进行连接操作。

16. 当不同的数字类型相加，系统自动执行强制类型转换。

17. 运算符 //执行地板除： // 除法不管操作数何种数值类型，总是舍去小数部分，返回数字序列中比真正的商小的最接近的数字。

```
>>> 1 // 2 # floors result, returns integer # 地板除， 返回整数
0
>>> 1.0 // 2.0 # floors result, returns float # 地板除， 返回浮点数
0.0
```

18. 用数字*x 一个序列会生成一个新序列，而在新序列中，原来有序列被重复 x 次。(乘、重复)

```
>>> 'python' * 5
'pythonpythonpythonpythonpython'
>>> [42] * 10
[42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

19. 因为字符串不能像列表一样被修改，所以有时根据字符串创建列表会很有用。list 函数可以实现这个操作：

```
>>> list('Hello')
['H','e','l','l','o']
```

注意，list 函数适用于所有类型的序列，而不只是字符串。

20. 在原始字符串里，所有的字符都是直接按照字面的意思来使用，没有转义特殊或不能打印的字符。引号前面加 r（大小写均可）表示原始字符串。

```
>>> print r'\n'
\n
```

这里指的是\n 而不是换行符。

21. Unicode 字符串操作符(u/U)，Unicode 操作符必须出现在原始字符串操作符前面。

```
u'Hello\nWorld!'
```

26.分片是一个非常强大的特性，分片赋值操作则更加显现它的强大。

```
>>> name = list('Perl')
>>> name
['P', 'e', 'r', 'l']
>>> name[2:] = list('ar')
>>> name
['P', 'e', 'a', 'r']
```

27.tuple 函数以一个序列作为参数并把它转换为元组。

```
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple((1, 2, 3))
(1, 2, 3)
```

28.当元组只有一个元素时需要加一个逗号，用以防止跟普通的分组操作符混淆。

```
a=('abc',)
print a[0]
```

如果不加逗号，输出的是“a”，加了逗号，输出的是“abc”。

29.浅拷贝事实上只拷贝了变量的引用（类似 C 语言的引用传递），由 copy 模块的 copy()实现；深拷贝就创建一个全新的对象（类似 C 语言的传值），由 decopy()函数实现。

30.dict()工厂函数被用来创建字典。如果不提供参数，会生成空字典。

```
>>> items = [('name', 'Gumby'), ('age', 42)]
>>> d = dict(items)
>>> d
{'age': 42, 'name': 'Gumby'}
>>> d['name']
'Gumby'
```

31.集合是一个无序不重复元素集，基本功能包括关系测试和消除重复元素。

- Java 中的集合包括 list、set 和 map 等，而 Python 中的集合只包括 set。

32.可以使用 set 解决海量列表里重复元素问题：

```
>>> a = [11,22,33,44,11,22]
>>> b = set(a)
>>> b
set([33, 11, 44, 22])
>>> c = [i for i in b]
>>> c
[33, 11, 44, 22]
```

33. 与列表和元组不同，集合是无序的，也无法通过数字进行索引。此外，集合中的元素不能重复。

34. set 不支持 indexing 和切片，但支持 len（set）、x in set 等操作。

35. 集合包括两种：

- set：可变集合，可以使用 add()、remove()等函数改变集合大小，用 set()函数生成。
- frozenset：不可变集合，一旦集合生成，就无法改变。用 frozenset()生成。

33. set 的 issubset()函数和“<=”都用于判断一个集合对象是否是另一个集合对象的子集：

```
s.issubset(t)
s <= t
```

测试是否 s 中的每一个元素都在 t 中。

34. set 的 issuperset()函数和“>=”都用于判断一个对象是否是另一个对象的超集：

```
s.issuperset(t)
s >= t
```

测试是否 t 中的每一个元素都在 s 中。

35. set()和 frozenset()工厂函数分别用来生成可变和不可变的集合。

36. 列表解析来自函数式编程语言(haskell)，语法如下：

```
[expr for iter_var in iterable]
[expr for iter_var in iterable if cond_expr]
```

第一种语法：首先迭代 iterable 里所有内容，每一次迭代，都把 iterable 里相应内容放到 iter_var 中，再在表达式中应用该 iter_var 的内容，最后用表达式的计算值生成一个列表。

第二种语法：加入了判断语句，只有满足条件的内容才把 iterable 里相应内容放到 iter_var 中，再在表达式中应用该 iter_var 的内容，最后用表达式的计算值生成一个列表。

```
all = []
fobj.writelines(['%s%s' % (x, ls) for x in all])#列表解析，将内容输入到文件中
```

37. sys 模块通过 sys.argv 属性提供了对命令行参数的访问：

- sys.argv 是命令行参数的列表
- len(sys.argv) 是命令行参数的个数(也就是 argc)

41. 永久存储模块应用场景：在简单的磁盘文件已经不能满足你的需要，而使用完整的关系数据库管理系统又有些大材小用。

- pickle 和 marshal 模块：用来转换并储存 Python 对象，并没有提供"永久性储存"的功能，因为它们没有为对象提供名称空间，也没有提供对永久性储存对象的并发写入访问。
- DBM 风格的模块：*db*系列的模块使用传统的 DBM 格式写入数据，Python 提供了 DBM 的多种实现：dbhash/bsddb，dbm，gdbm，以及 dumbdbm 等。这些模块为用户的对象提供了一个命名空间，这些对象同时具备字典对象和文件对象的特点。不过不足之处在于它们只能储存字符串，不能对 Python 对象进行序列化。
- shelve 模块：这是永久存储模块更为完整的解决方案。

43. try 语句有两种主要形式：try-except 和 try-finally。一个 try 语句可以对应一个或多个 except 子句, 但只能对应一个 finally 子句。

44. 最常见的 try-except 语句语法如下所示。它由 try 块和 except 块组成，也可以有一个可选的错误原因：

```
try:
    try_suite # watch for exceptions here 监控这里的异常
except Exception[, reason]:
    except_suite # exception-handling code 异常处理代码
```

例：

```
>>> try:
...     f = open('blah', 'r')
... except IOError, e:
...     print 'could not open file:', e
...
could not open file: [Errno 2] No such file or directory
```

当引发 IOError 异常时, 我们告诉解释器让它打印出一条诊断信息。

45. 异常中的 else 子句：在 try 范围中没有异常被检测到时，执行 else 子句：

```
try:
```

```
    3rd_party_module.function()
except:
    log.write("*** caught exception in module\n")
else:
    log.write("*** no exceptions caught\n")
```

46. finally 子句是无论异常是否发生，是否捕捉都会执行的一段代码。你可以将 finally 仅仅配合 try 一起使用，也可以和 try-except(else 也是可选的)一起使用。

```
try:
    A
    except MyException: B
else: C
finally: D
```

47. try-finally 语句和 try-except 区别在于它不是用来捕捉异常的。作为替代，它常常用来维持一致的行为而无论异常是否发生。我们得知无论 try 中是否有异常触发，finally 代码段都会被执行。

```
try:
    try_suite
finally:
    finally_suite #无论如何都执行
```

例如：

```
try:
   f = open('xxx')
except:
   print 'fail to open'
   exit(-1)
try:
   do something
except:
   do something
finally:
   f.close()                #无论 try 中是否有异常发生，都必须关闭文件
```

当在 try 范围中产生一个异常时，(这里)会立即跳转到 finally 语句段。当 finally 中的所有代码都执行完毕后，会继续向上一层引发异常。

48. with 语句也是用于 try-finally 代码，with 所请求的对象必须有一个__enter__()方法，一个__exit__()方法。__exit__()其实类似于 finally 语句，不管处理过程中是否出现错误，__exit__()语句都会被调用。

- __enter__()：和__init__()方法类似，用于在正式执行前的初始化操作。要注意的是，__enter__()方法的返回值会赋值给 as 后面的变量，而并非是紧跟在 with 后面的那个函数或类。例如：

```
with open('/Users/Mr.Long/Desktop/data.txt', 'w') as f:
    f.write('hello world')
```

在执行写操作前，该文件对象的__enter__()会被调用，__enter__()的返回值赋值给 f，而不是 open()函数的返回值赋值给 f。

- __exit__()：操作执行完后的清理操作。在上面的写操作执行完后，不管是否出错，都会调用文件对象的__exit__()操作来进行清理，例如释放存储空间等。

```
#coding=utf-8

class Sample:
  def __enter__(self):
    print "正在 enter 函数"
     return self

  def __exit__(self, *args):
    print "退出 exit 函数"

  def do_something(self):
    print "正在处理"

with Sample() as sample:
  sample.do_something()
```

输出为：

```
正在 enter 函数
正在处理
退出 exit 函数
```

49. raise 语句提供了一种机制让程序员明确的触发异常。rasie 一般的用法是：

```
raise [SomeException [, args [, traceback]]]
```

50.断言是一句必须等价于布尔真的判定；此外，发生异常也意味着表达式为假。测试一个表达式，如果返回值是假，触发异常，程序崩溃。如果需要确保程序中的某个条件一定为真才能让程序正常工作的话，assert 语句就有用了，它可以在程序中置入检查点。断言通过 assert 语句实现：

```
assert expression[, arguments]
```

13.内建的 zip 函数可以用来进行并行迭代，可以把两个序列“压缩”在一起，然后返回一个元组的列表：

```
names = ['anne', 'beth', 'george', 'damon']
ages = [12, 45, 32, 102]
```

如下所示：

```
>>> zip(names, ages)
[('anne', 12), ('beth', 45), ('george', 32), ('damon', 102)]
```

第 11 章 函数和函数式编程

1. 和其他高级语言类似，Python 也不允许在函数未声明之前，对其进行引用或者调用。

2. 函数也有属性：

```
def f1():
……
clo = f1.func_closure #func_closure: 这个属性仅当函数是一个闭包时有效，指向一个保存了
                # 所引用到的外部函数的变量 cell 的元组。这个属性也是只读的。
```

3. python 允许用 lambda 关键字创造匿名函数。lambda 是为了减少单行函数的定义而存在的一个完整的。 lambda“语句”代表了一个表达式，这个表达式的定义体必须和声明放在同一行。冒号前面的是入口参数。我们现在来演示下匿名函数的语法：

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

为了调用这个函数，我们将它保存到一个变量中，以后可以随时调用。例子：

```
>>> a = lambda x, y=2: x + y
>>> a(3)
5
>>> a(3,5)
8
>>> a(0)
```

```
2
>>> a(0,9)
9
>>>
```

4. python 变量可以指向函数，函数名也是变量。

5. 高阶函数就是可以接收另一个函数作为参数的函数：

```
def add(x, y, f):
  return f(x) + f(y)
```

其中 f 是函数。

- 其实就相当于调用回调函数。

6. global 语句明确地引用一个已命名的全局变量，global 的语法如下：

```
global var1[, var2[, ... varN]]]
```

7. 装饰器用于增强函数的功能：

(1) 不带参数：

```
def log(func):
  def wrapper(*args, **kw):
    print 'call %s():' % func.__name__
    return func(*args, **kw)
  return wrapper
```

log()接受一个函数作为参数，然后返回一个函数。

```
@log
def now():
  print '2013-12-25'
```

这时调用 now()函数，不仅会运行 now()函数本身，还会在运行 now()函数前执行 wrapper()打印一行日志：

```
>>> now()
call now():
2013-12-25
```

- 第一个 return 返回的是下一层的函数名，第二个 return 返回的是外层函数的参数。
- 不带参数的装饰器是 2 层的函数，类似于：

```
def AA(func):
```

```
    def BB(*args, **kw):
        ......
        return func(*args, **kw)
    return BB
```

而带参数的装饰器比不带参数的装饰器多一层，共 3 层：

```
def AA(text):
    def BB(func):
        def CC(*args, **kw):
            ......
            return func(*args, **kw)
        return CC
    return BB
```

(2) 如果需要传入参数，则编写一个高阶函数：

```
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print '%s %s():' % (text, func.__name__)
            return func(*args, **kw)
        return wrapper
    return decorator
```

decorator 是一个高阶函数。

```
@log('execute')
def now():
    print '2013-12-25'
```

执行结果如下：

```
>>> now()
execute now():
2013-12-25
```

8．装饰器其实就是闭包的一种应用。

9．列表解析直接创建列表，当列表有大量数据时，就会出现内存浪费的现象。为了解决这个问题，引入了生成器。生成器作用的列表解析类似，只是，生成器是按给定的算法推算出来，推算一个就使用一个，而列表解析是一次性生成所有数据。

10. 生成器有两种：

- 生成器表达式： 语法和列表解析类似，只不过把列表解析的中括号[]换成括号()。

```
>>> gen = (x**2 for x in range(5))
>>> gen
<generator object <genexpr> at 0x0000000002FB7B40>  # generator object 指生成器对象
>>> for g in gen:
...   print(g, end='-')
...
0-1-4-9-16-
>>> for x in [0,1,2,3,4,5]:
...   print(x, end='-')
...
0-1-2-3-4-5-
```

- 生成器函数： 在函数中如果出现了 yield 关键字，那么该函数就不再是普通函数，而是生成器函数。

11. 生成器函数是一个带 yield 语句的函数。yield 语句的功能是返回一个值给调用者并暂停执行。当生成器的 next()方法被调用的时候，它会准确地从离开的地方继续执行。它记住上一次返回时在函数体中的位置。对生成器函数的第二次（或第 n 次）调用跳转至该函数中间，而上次调用的所有局部变量都保持不变。生成器不仅“记住”了它数据状态；生成器还“记住”了它在流控制构造（在命令式编程中，这种构造不只是数据值）中的位置。

```
nested = [[1, 2], [3. 4], [5]]

def flatten(nested):
    for sublist in nested:
        for element in sublist:
            yield element            #在这里暂时并返回 element，下次调用时继续执行 for 语句
```

调用生成器函数将返回一个生成器对象：

```
>>> generator = flatten()
>>> generator.next()
```

此时 genreator 是个生成器，也是个函数。

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，并且 yield 的参数将作为此次 next 方法的返回值：

```
>>> generator.next()
```

之后每次调用生成器的 next 方法，生成器将从上次暂停执行的位置恢复执行生成器函数，直到再次遇到 yield 时暂停，并且同样的，yield 的参数将作为 next 方法的返回值。

说明：当你问生成器要一个数时，生成器会执行，直至出现 yield 语句，生成器把 yield 的参数给你，之后生成器就不会往下继续运行。 当你问他要下一个数时，他会从上次的状态。开始运行，直至出现 yield 语句，把参数给你，之后停下。如此反复直至退出函数。

12.生成器方法：

- 外部作用域访问生成器的 send 方法，就像访问 next 方法一样，只不过前者使用一个参数(要发送的“消息”—任意对象)。当生成器重新运行的时候，yield 方法返回一个值，也就是外部通过 send 方法发送的值。如果 next 方法被使用，那么 yield 方法返 None。注意，使用 send 方法只有在生成器挂起之后才有意义。
- throw 方法用于在生成器内引发一个异常。
- close 方法用于停止生成器。

第 12 章 模块

1. 在 Python 中，每个 Python 文件都可以作为一个模块，模块的名字就是文件的名字。比如有这样一个文件 test.py，在 test.py 中定义了函数 add：

```
#test.py

def add(a,b):
  return a+b
```

那么在其他文件中就可以先 import test，然后通过 test.add(a,b)来调用了，当然也可以通过 from test import add 来引入。

2. 模块的_file_ 属性保存着模块文件源码的路径。

3. 模块：

- sys 这个模块让你能够访问与 Python 解释器联系紧密的变量和函数。

表 10-2 sys模块中一些重要的函数和变量

函数/变量	描 述
argv	命令行参数，包括脚本名称
exit([arg])	退出当前的程序，可选参数为给定的返回值或者错误信息
modules	映射模块名字到载入模块的字典
path	查找模块所在目录的目录名列表
platform	类似sunos5或者win32的平台标识符
stdin	标准输入流——一个类文件（file-like）对象
stdout	标准输出流——一个类文件对象
stderr	标准错误流——一个类文件对象

- OS 模块为你提供了访问多个操作系统服务的功能。
- fileinput 模块让你能够轻松地遍历文本文件的所有行。
- time 模块所包括的函数能够实现以下功能：获得当前时间、操作时间和日期、从字符串读取时间以及格式化时间为字符串。
- random 模块包括返回随机数的函数，可以用于模拟或者用于任何产生随机输出的程序。

4. 你可以在你的模块里导入指定的模块属性。 也就是把指定名称导入到当前作用域。使用 from-import 语句可以实现我们的目的，它的语法是：

```
from module import name1[, name2[,... nameN]]
from Tkinter import Tk, Frame, Button, Entry, Canvas, \          #多行导入
     Text, LEFT, DISABLED, NORMAL, RIDGE, END
```

5. 使用自己想要的名字替换模块的原始名称：

```
import Tkinter as tk
```

6. 一个模块只被加载一次，无论它被导入多少次。
7. 调用 from-import 可以把名字导入当前的名称空间里去，这意味着你不需要使用属性/句点属性标识来访问模块的标识符。
8. __import__() 函数作为实际上导入模块的函数，这意味着 import 语句调用 __import__() 函数完成它的工作。提供这个函数是为了让有特殊需要的用户覆盖它，实现自定义的导入算法。

```
__import__(module_name[, globals[, locals[, fromlist]]])
```

9. globals() 和 locals() 内建函数分别返回调用者全局和局部名称空间的字典。
10. reload() 内建函数可以重新导入一个已经导入的模块。 它的语法如下：

```
reload(module_name)
```

第 13 章 面向对象编程

1. Python 类使用 class 关键字来创建。简单的类的声明可以是关键字后紧跟类名，如下所示：

```
class ClassName(bases):              #base 是基类
    'class documentation string'    #类文档字符串
    class_suite                          #类体
```

按照 Python 的编程习惯，类名以大写字母开头，紧接着是(object)，表示该类是从哪个类继承下来的。创建实例使用 类名+()，类似函数调用的形式创建：

```
xiaoming = Person()
xiaohong = Person()
```

2. Python 并不支持纯虚函数。

3. 很多其它的 OO 语言都提供 new 关键字，通过 new 可以创建类的实例。Python 的方式更加简单。一旦定义了一个类，创建实例比调用一个函数还容易------不费吹灰之力。实例化的实现，可以使用函数操作符，如下示：

```
>>> class MyClass(object): # define class 定义类
... pass
>>> mc = MyClass() # instantiate class 初始化类
```

4. 类的专有方法需要自己去实现，以实现特殊的功能，如定义__len__函数可以让类可以在类外部使用 len()函数来返回对象的大小。专有方法有如下几个：

(1) __init__(self,...)：构造函数，这个方法在新建对象要被返回使用之前被调用。

(2) __del__(self)：析构函数，在对象要被删除之前调用。

(3) __str__(self)：在我们对对象使用 print 语句或是使用 str()的时候调用。

(4) __lt__(self,other)：当使用 小于 运算符（<）的时候调用。类似地，对于所有的运算符（+，>等等）都有特殊的方法。

(5) __getitem__(self,key)：使用 x[key]索引操作符的时候调用。

(6) __len__(self)：对序列对象使用内建的 len()函数的时候调用。

示例：

```
class MySeq:

    def __init__(self):
        self.lseq = ["I","II","III","IV"]
    def __len__(self):
        return len(self.lseq)
```

```
    def __getitem__(self,key):
        if 0 <= key < 4:
        return self.lseq[key]

    if __name__ == '__main__':
        m = MySeq()
        for i in range(4):
        print(m[i])
```

5. 在定义类时，可以为类添加一个特殊的__init__()方法，当创建实例时，__init__()方法被自动调用。__init__() 方法的第一个参数必须是 self，后续参数则可以自由指定，和定义函数没有任何区别。

```
class Person(object):
  def __init__(self, name, gender, birth):
    self.name = name
    self.gender = gender
    self.birth = birth
```

相应地，创建实例时，就必须要提供除 self 以外的参数：

```
xiaoming = Person('Xiao Ming', 'Male', '1991-1-1')
xiaohong = Person('Xiao Hong', 'Female', '1992-2-2')
```

6. __del__()类似 C++中的析构函数。
7. 以"__xxx__"定义的属性在 Python 的类中被称为特殊属性，通常不要把普通属性用"__xxx__"定义。
8. 一个实例的私有属性就是以__开头的属性，无法被外部访问。
9. 实例的方法就是在类中定义的函数，它的第一个参数永远是 self。指向调用该方法的实例本身，其他参数和一个普通函数是完全一样的。方法本身也是属性。

```
class Person(object):

  def __init__(self, name):
    self.__name = name

  def get_name(self):
    return self.__name
```

10.如果要把一个类的实例变成 str，就需要实现特殊方法__str__()：

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
    def __str__(self):
        return '(Person: %s, %s)' % (self.name, self.gender)
```

现在，在交互式命令行下用 print 试试：

```
>>> p = Person('Bob', 'male')
>>> print p
(Person: Bob, male)
```

但是，如果直接敲变量 p：

```
>>> p
<main.Person object at 0x10c941890>
```

Python 定义了__str__()和__repr__()两种方法，__str__()用于显示给用户，而__repr__()用于显示给开发人员。

11.对 int、str 等内置数据类型排序时，Python 的 sorted() 按照默认的比较函数 cmp 排序，但是，如果对一组 Student 类的实例排序时，就必须提供我们自己的特殊方法 __cmp__()：

```
class Student(object):
    ……
    def __cmp__(self, s):
        if self.name < s.name:
            return -1
        elif self.name > s.name:
            return 1
        else:
            return 0
```

12.如果一个类表现得像一个 list，要获取有多少个元素，就得用 len() 函数。要让 len() 函数工作正常，类必须提供一个特殊方法__len__()，它返回元素的个数。

```
class Students(object):
    def __init__(self, *args):
        self.names = args
    def __len__(self):
        return len(self.names)
```

13. 创建子类的语法看起来与普通(新式)类没有区别， 一个类名，后跟一个或多个需要从其中派生的父类：

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
    'optional class documentation string'
    class_suite
```

如果你的类没有从任何祖先类派生，可以使用 object 作为父类的名字。经典类的声明唯一不同之处在于其没有从祖先类派生－－－此时，没有圆括号：

```
class ClassicClassWithoutSuperclasses:
    pass
```

14. __bases__类属性列出已知类的基类。

15. 继承时重写__init__不会自动调用基类的__init__。

16. 同 C++一样，Python 允许子类继承多个基类。

```
class 类名(父类 1,父类 2,....,父类 n):
  <语句 1>
```

17. issubclass() 布尔函数判断一个类是另一个类的子类或子孙类。

18. 函数 isinstance()可以判断一个变量的类型，既可以用在 Python 内置的数据类型如 str、list、dict，也可以用在我们自定义的类，它们本质上都是数据类型。

19. hasattr()、getattr()、setattr()、delattr()函数可以在各种对象下工作，不限于类（class）和实例（instances）。

20. 默认情况下，属性在 Python 中都是“public”，类所在模块和导入了类所在模块的其他模块的代码都可以访问到。

第 14 章 执行环境

1. 当模块导入后，就执行所有的模块。

2. os.system()接收字符串形式的系统命令并执行它。

3. os.popen()工作方式和 system()相同，但它可以建立一个指向那个程序的单向连接，然后如访问文件一样访问这个程序。(os.popen()、单向连接)

第 16 章 网络编程

1. 要使用 socket.socket()函数来创建套接字。其语法如下：

```
socket(socket_family, socket_type, protocol=0)
```

2. 创建一个 TCP 服务器伪代码：

```
ss = socket() # 创建服务器套接字
ss.bind() # 把地址绑定到套接字上
ss.listen() # 监听连接
inf_loop: # 服务器无限循环
cs = ss.accept() # 接受客户的连接
comm_loop: # 通讯循环
cs.recv()/cs.send() # 对话（接收与发送）
cs.close() # 关闭客户套接字
ss.close() # 关闭服务器套接字（可选）
```

3. 创建 TCP 客户端伪代码：

```
cs = socket() # 创建客户套接字
cs.connect() # 尝试连接服务器
comm_loop: # 通讯循环
cs.send()/cs.recv() # 对话（发送／接收）
cs.close() # 关闭客户套接字
```

4. 创建一个 UDP 服务器

```
ss = socket() # 创建一个服务器套接字
ss.bind() # 绑定服务器套接字
inf_loop: # 服务器无限循环
cs = ss.recvfrom()/ss.sendto() # 对话（接收与发送）
ss.close() # 关闭服务器套接字
```

5. 创建一个 UDP 客户端

```
cs = socket() # 创建客户套接字
comm_loop: # 通讯循环
cs.sendto()/cs.recvfrom() # 对话（发送／接收）
cs.close() # 关闭客户套接字
```

6. SocketServer 是标准库中一个高级别的模块。用于简化网络客户与服务器的实现。模块中，已经实现了一些可供使用的类。
7. Twisted 是一个完全事件驱动的网络框架。 它允许你使用和开发完全异步的网络应用程序和协议。

第 17 章 网络客户端编程

1. 在使用 Python 的 FTP 支持时，你所需要做的就是导入 ftplib 模块，并实例化一个 ftplib.FTP 类对象。所有的 FTP 操作 都要使用这个对象来完成。 下面是一段 Python 的伪代码：

```
from ftplib import FTP
f = FTP('ftp.python.org')
f.login('anonymous', 'guess@who.org')
:
f.quit()
```

第 18 章 多线程编程

1. Python 代码的执行由 Python 虚拟机(也叫解释器主循环)来控制。对 Python 虚拟机的访问由全局解释器锁（GIL）来控制，正是这个锁能保证同一时刻只有一个线程在运行。
2. thread 模块，当主线程退出的时候，所有其它线程没有被清除就退出了。threading 模块能确保所有“重要的”子线程都退出后，进程才会结束。避免使用 thread 模块。(threading、确保所有线程退出、才结束)
3. Python 提供了几个用于多线程编程的模块，包括 thread, threading 和 Queue 等。thread 和 threading 模块允许程序员创建和管理线程。thread 模块提供了基本的线程和锁的支持，而 threading 提供了更高级别，功能更强的线程管理的功能。Queue 模块允许用户创建一个可以用于多个线程之间共享数据的队列数据结构。
4. threading 的 Thread 类是你主要的运行对象。它有很多 thread 模块里没有的函数。

第 20 章 Web 编程

1. 浏览器只是 Web 客户端的一种。任何一个通过向服务器端发送请求来获得数据的应用程序都被认为是“客户端”。

2. Python 支持两种不同的模块，分别以不同的功能和兼容性来处理 URL。一种是 urlparse，一种是 urllib。

- urlpasrse 模块提供了操作 URL 字符串的基本功能。这些功能包括 urlparse(), urlunparse()和 urljoin()。
- urlparse()将 URL 字符串拆分成网络协议或者下载规划、服务器位置(或许也有用户信息)等部件。

3. urllib 提供了一个高级的 Web 交流库，支持 Web 协议， HTTP， FTP 和 Gopher 协议，同时也支持对本地文件的访问。
4. 基础的(Web)服务器是一个必备的模具。它的角色是在客户端和服务器端完成必要 HTTP 交互。在 BaseHTTPServer 模块中你可以找到一个名叫 HTTPServer 的服务器基本类。
5. 处理器是一些处理主要“Web 服务”的简单软件。它们处理客户端的请求，并返回适当的文件，静态的文本或者由 CGI 生成的动态文件。 处理器的复杂性决定了你的 Web 服务器的复杂程度。 Python 标准库提供了三种不同的处理器:

- 最基本，最普通的是 vanilla 处理器，被命名 BaseHTTPResquestHandler，这个可以在基本 Web 服务器的 BaseHTTPServer 模块中找到。
- 用 于 SimpleHTTPServer 模 块 中 的 SimpleHTTPRequestHandler ， 建 立在 BaseHTTPResquestHandler 基础上，直接执行标准的 GET 和 HEAD 请求。
- 最后，我们来看下用于 CGIHTTPServer 模块中的 CGIHTTPRequestHandler 处理器，它可以获取 SimpleHTTPRequestHandler 并为 POST 请求提供支持。它可以调用 CGI 脚本完成请求处理过程，也可以将生成的 HTML 脚本返回给客户端。

第 21 章 数据库编程

1. DB-API 是一个规范，它定义了一系列必须的对象和数据库存取方式，以便为各种各样的底层数据库系统和多种多样的数据库接口程序提供一致的访问接口。

   说明：不同的数据库拥有不同的接口程序，如 MySQL 的接口程序为 MySQLdb。

2. ORM： 将绝大多数纯 SQL 层功能抽象为 Python 对象, 这样你就无需编写 SQL 也能够完成同样的任务。

第 22 章 日志模块

1. Python 的 logging 模块提供了通用的日志系统，可以方便第三方模块或者是应用使用。模块提供了许多组件：记录器、处理器、过滤器和格式化器。

- logger：提供日志接口，供应用代码使用。logger 最长用的操作有两类：配置和发送日志消息。可以通过 logging.getLogger(name)获取 logger 对象，如果不指定 name 则返回 root 对象，多次使用相同的 name 调用 getLogger 方法返回同一个 logger 对象。
- handler：将日志记录（log record）发送到合适的目的地（destination），比如文件，socket 等。一个 logger 对象可以通过 addHandler 方法添加 0 到多个 handler，每个 handler 又可以定义不同日志级别，以实现日志分级过滤显示。
- filter：提供一种优雅的方式决定一个日志记录是否发送到 handler。
- formatter：指定日志记录输出的具体格式。formatter 的构造方法需要两个参数：消息的格式字符串和日期字符串，这两个参数都是可选的。

2. 常用的记录器对象的方法分为两类：配置和发送消息。最常用的配置方法：

- Logger.setLevel()指明了记录器能处理的日志消息的最低级别，内置级别中 debug（调试）最低，critical（严重）最高。例如，如果严重程度为 INFO，记录器将只处理 INFO，WARNING，ERROR 和 CRITICAL 消息，DEBUG 消息被忽略。
- Logger.addHandler() 和 Logger.removeHandler() 在记录器对象上添加或者移除处理器对象。
- Logger.addFilter() 和 Logger.removeFilter() 在记录器对象上添加或者移除过滤器对象。

记录器对象配置好了后，下面的方法可以用来创建日志消息：

- Logger.debug()，Logger.info()，Logger.warning()，Logger.error()和 Logger.critical()创建带消息的日志记录，记录的级别为对应的方法名。
- Logger.exception()创建类似于 Logger.error()的日志消息。
- Logger.log()以日志级别为显式参数。

3. Handler／处理器对象用来将合适的日志消息分发到处理器特定的目的地（基于日志消息的级别）。举个例子，一个应用可以将所有的日志消息发送至日志文件，所有的错误（error）及其以上的日志消息发送至标准输出，所有的严重的（critical）日志消息发送至某个电子邮箱。

附录 A

1. python 是脚本语言，脚本从头执行到尾，并没有入口函数，而 C 和 C++语言有入口函数 main，C 语言循环只能使用 while，而 C++有相应的机制，比如 QT 有 exec。

2. else 子句既可以与 if 一起使用，也可以与 for 一起使用，还可以与 while 一起使用。

3. 脚本被执行的时候，__name__ 值就是 __main__ ，才会执行 main()函数，如果这个脚本是被 import 的话，__name__的值通常为模块文件名，不带路径或者文件扩展名。main()函数就不会被调用。

```
if __name__ == '__main__':
   main()
```

也就是说，该脚本不能作为模块导入。

4. 函数

- choice()用于随机获取参数中的一个值。
- randint()用于生成随机数。

5. 注意下面这种用法：

```
ops = {'+': add, '-': sub}        #定义字典
op = choice('+-')              #随机获取一个符号
nums = [ randint(1,10) for i in range(2) ]      #随机获取 2 个 0 到 9 的数
nums.sort(reverse=True)                  #降序排序
ans = ops[op](*nums)        #注意这种用法，这里应该指的是 add()和 sub()
```

6. 注意三引号这种用法：

```
print """Called:
function: %s
args: %r
kargs: %r""" % (f, args, kargs)
```

7. functools.partial 可以为指定函数指定一个默认参数

8. 下面这条语句可以输出 20 个‘-’

```
print '-' * 20                  #输出 20 个-号
```

9. Python 中的列表(list)保存一组值，由于列表的元素可以是任何对象，比较浪费内存；而 Python 的 array 模块不支持多维，也没有各种运算函数，因此也不适合做数值运算。NumPy 的诞生弥补了这些不足，NumPy 提供了两种基本的对象：ndarray 和 ufunc。ndarray(下文统一称之为数组)是存储单一数据类型的多维数组，而 ufunc 则是能够对数组进行处理的函数。

10. pickle 模块的 dump 函数用于将对象转化为文件保存到磁盘上，load 函数将文件还原。

- pickle.dump(obj, file[, protocol])

这是将对象持久化的方法，参数的含义分别为：

- pickle.load(file)

只有一个参数 file ，对应于上面 dump 方法中的 file 参数。

12. 使用 threading.Event 可以使一个线程等待其他线程的通知，我们把这个 Event 传递到线程对象中，Event 默认内置了一个标志，初始值为 False。一旦该线程通过 wait()方法进入等待状态，直到另一个线程调用该 Event 的 set()方法将内置标志设置为 True 时，该 Event 会通知所有等待状态的线程恢复运行。

13. python 弱化了类型的作用。

14. python 的 os 模块很多函数名和 unix 下的系统调用一样的，如 os.setuid 和系统调用 setuid。

15. 标准库中有很多函数参数是 Iterable 对象（可迭代对象），注意和迭代器 Iterator 的区别：Python 中 list，truple，str，dict 这些都可以被迭代，是可迭代对象，但它们不是迭代器。可以用可以 for 循环迭代的，都是 Iterable；相反，用 next()进行迭代的，都是 Iterator。如何判断一个对象是否可迭代？通过 collections 模块的 Iterable 类型判断：

```
>>> from collections import Iterable
>>> isinstance('abc', Iterable) # str 是否可迭代
True
>>> isinstance([1,2,3], Iterable) # list 是否可迭代
True
>>> isinstance(123, Iterable) # 整数是否可迭代
False
```

16. 正则表达式中，特殊字符需要进行转义，即在其前加一个\。特殊字符包括：$、( )、*、+、.、[、?、\、^、{、|。注意，中括号和大括号只有半边是特殊字符，小括号两边都是。

17. 正则表达式中，*?、+?、??、{m,n}?　前面的*、+等都是贪婪匹配，也就是尽可能匹配，后面加?号使其变成惰性匹配，例如匹配网页文件中的 href 标签：

```
href=(.*)                贪婪匹配，会匹配到大量非 href 标签内容
href=(.*?)               惰性匹配，只会匹配 href 标签
```

18. 正则表达式中，可以分为查找和提取两项操作

- 查找：不加括号的默认就是查找操作

```
find = re.compile(r"<a class='pagebar_pages'  href='javascript:goPage\
(.*?\)'>.*?<\/a> ")
```

默认返回整个 a 标签。

- 提取；加括号以后，正则表达式会提取括号中的内容。

```
find = re.compile(r"<a class='pagebar_pages'  href='javascript:goPage\(.*?\)'>(.*?)<\/
a> ")
```

这个正则表达式只会返回第二个括号的内容

19. Python 内置的@property 装饰器负责把一个方法变成属性调用，它可以有 getter 方法和 setter 方法。这样做的意义是为了防止直接暴露给外部导致数值不可控。实际上就是使用该方法获取并返回一个属性。

```
class Cls(object):
    def __init__(self):
        self.__x = None

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, value):
        self.__x = value

    @x.deleter
    def x(self):
        del self.__x

if __name__ == '__main__':
    c = Cls()
    c.x = 100
    y = c.x
    print("set & get y: %d" % y)

    del c.x
    print("del c.x & y: %d" % y)
```

@property 定义的属性和普通属性的使用方法是一样的。如上所述，虽然 x 是按函数方式定义的，但使用方式是“c.x”，而不是“c.x()”。

20. python 内部默认使用 unicode 码，可以通过 coding 语句定义默认编码。python 给变量赋值时，前面带 u 的表示该变量是 unicode 码，不带 u 的表示该变量是个字符串，脚本设置的默认编码，由 unicode 经过编码(encode)后的字节组成。

```
>>> #coding=utf-8
... u=u'汉字'              #u 是 unicode
>>> type(u)
<type 'unicode'>
>>> s='汉字'              #i 是字符串， utf-8 编码，由 unicode 经过编码(encode)后的字节组成
>>> type(s)
<type 'str'>
>>> type(s)
```

unicode 是一种字符集，unicode 码就是直接使用 unicode 字符集来保存数据；而 utf-8 和 gbk 等是一种 unicode 编码规则，也就是使用 unicode 字符集的变种来保存数据。

21. 任何两个非 unicode 之间（如 gbk 和 utf-8 之间）使用 unicode 码为中间码。

22. encode()是将字符串编码成括号中的编码集，而 decode 则是使用括号中的编码将字符串解码成 unicode 码。

23. 在 utf-8 中，汉字需要将 utf-8 编码解码成 unicode 码才能正确显示：

```
#coding=utf-8
s = '中文'                              #这个是 utf-8 码，所以这个会出现乱码
print s
s = '中文'.decode('utf-8')          #这个已经将 utf-8 解码成 unicode 码，正常显示
print s
s = u'中文'                            #本来就是 unicode 码
print s
```

输出为：

```
E:\Download>python test.py
涓鏃构
中文
中文
```

附录 B Java 语言、python 和 shell 语法的区别

1. Python 和 Shell 语句结束时不需要加分号，而 Java 需要加分号。
2. Python 中主要的数据结构有字符串、元组、列表和字典；

而 Java 中主要的数据结构有字符串、List 接口及其实现、Map 接口及其实现、Set 接口及其实现、Vector 类及其子类 Stack 类等。

Shell 中只有字符串和数组两种数组结构。

3. Python 中和 shell 定义变量都不需要声明变量类型，直接定义即可，而 Java 中需要声明变量的类型。

- Java：

```
Int num=20;
String str="hello";
```

- Python 和 shell：

```
counter = 100 # 赋值整型变量
miles = 1000.0 # 浮点型
name = "John" # 字符串
```

4. Python 和 shell 中的数据结构都可以直接赋值，而 Java 中数据结构有些可以直接赋值，例如数组和字符串，有些需要使用 new 来赋值，例如 list 和 map。

- Java：

```
int arr[]=new int[]{1,2,3,4,5};              //一维数组的第一种初始化方式
int arr2[]={4,5,6,7,8};                      //一维数组的第二种初始化方式，这是简化形式
String str="字符串常量";                      //字符串的第一种初始化方式
String s2=new String(“kvill”);               //字符串的第二种初始化方式
```

- Python：

```
var1 = 'Hello World!'                    ＃字符串
list1 = ['physics', 'chemistry', 1997, 2000];          ＃列表
tup1 = ('physics', 'chemistry', 1997, 2000);           ＃元组
```

- shell：

```
var2＝'Hello World!'
```

```
mytest=(one two three four)
```

5. if 语句：

- Java 语言的 if 语句语法为：

```
if(expression){
   expr_true_suite
}
else{
   expr_true_suite
}
```

- python 的 if 语句语法为：

```
if expression:
      expr_true_suite
else:
      expr_true_suite
```

- shell 的 if 语句语法为

```
if command
then
      commands
elif
      commands
else
      commands
fi
```

6. while 语句：

- Java 语言的 while 语句语法为：

```
while(condition) {
   statement(s);
}
```

- python 的 while 语句语法为：

```
while condition:
        statement
```

- shell 的 while 语句语法为：

```
while test command
do
    other commands
done
```

7. for 语句：

- Java 语言的 for 语句语法为：

```
for (表达式 1;表达式 2;表达式 3){
 循环语句
}
```

- python 的 for 语句语法为：

```
for iterating_var in sequence:
    statements(s)
```

- shell 的 for 语句语法为：

```
for var in list
do
    commands
done
```

8. Java 中的函数无法脱离类独立存在，任何一个函数都需要依附在一个类中，而 Python 中的类可以脱离类独立存在，shell 中则没有类的概念：

- Java 语言函数语法形如：

```
public class Yun{
void a(){
      //我是非静态方法
}
```

- python 函数语法：

```
def function_name([arguments]):          # def 是关键字
    function_suite
```

- shell 函数语法：

```
Function name{
    commands
}
```

或者：

```
Name() {
    commands
}
```

9. Python 中类的定义和继承都很像普通函数，其没有关键字 public 和 private，Python 把父类写到括号中；而 Java 中则有关键字 public 和 private，其继承使用的是 extends 关键字。

- Python：

```
#!/usr/bin/python
#coding=utf-8
class Parent:                            # 定义父类
    def myMethod(self):

class Child(Parent):                     # 定义子类
    def myMethod(self):
        print '调用子类方法'
```

- Java：

```
public class Animal {
    ……
}

public class Mouse extends Animal {
    ……
}
```

10. Java 使用 catch 语句来捕捉异常，catch 语句还带括号；Python 使用 except 语句来捕捉异常，except 语句不带括号：

- Java：

```
try{
        code that might throw exceptions
}
catch (FileNotFoundException e){
        emergency action for missing files
}
catch (UnknownHostException e){
        emergency action for unknown hosts
}
```

- Python：

```
try:
        fh = open("testfile", "w")
        fh.write("这是一个测试文件,用于测试异常!!")
except IOError:
        print "Error: 没有找到文件或读取文件失败"
else:
        print "内容写入文件成功"
        fh.close()
```

11.

附录 C 生产者模式

1. 在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。
2. 在线程方式下，生产者和消费者各自是一个线程。生产者把数据写入队列头（以下简称 push），消费者从队列尾部读出数据（以下简称 pop）。当队列为空，消费者就稍息（稍事休息）；当队列满（达到最大长度），生产者就稍息。整个流程并不复杂。

3. 由于两个线程共用一个队列，自然就会涉及到线程间诸如同步、互斥。
4. 跨进程的生产者／消费者模式，非常依赖于具体的进程间通讯（IPC）方式。

Celery

1. 异步任务是 web 开发中一个很常见的方法。对于一些耗时耗资源的操作，往往从主应用中隔离，通过异步的方式执行。
2. broker 是一个消息传输的中间件。每当应用程序调用 celery 的异步任务的时候，会向 broker 传递消息，而后 celery 的 worker 将会取到消息执行。
3. 通常程序发送的消息，发完就完了，可能都不知道对方什么时候接受了。为此，celery 实现了一个 backend，用于存储这些消息以及 celery 执行的一些消息和结果。对于 brokers，官方推荐是 rabbitmq 和 redis，至于 backend，就是数据库啦。
4. Celery 是 Python 开发的分布式任务调度模块。Celery 本身不含消息服务，它使用第三方消息服务来传递任务，目前，Celery 支持的消息服务有 RabbitMQ、Redis 甚至是数据库，当然 Redis 应该是最佳选择。
5. 使用 celery 包含三个方面，其一是定义任务函数，其二是运行 celery 服务，最后是客户应用程序的调用。
6. 使用 Redis 作为 Broker 时，再要安装一个 celery-with-redis。
7. Celery 使用方法：

(1) 编写 tasks.py

```
# tasks.py
import time
from celery import Celery

celery = Celery('tasks', broker='redis://localhost:6379/0')

@celery.task
def sendmail(mail):                    #这里是要异步执行的动作
   print('sending mail to %s...' % mail['to'])
   time.sleep(2.0)
   print('mail sent.')
```

(2) 然后启动 Celery 处理任务：

```
$ celery -A tasks worker --loglevel=info
```

(3) 客户端程序调用 tasks.py 里面的函数：

```
>>> from tasks import sendmail
>>> sendmail.delay(dict(to='celery@python.org'))
<AsyncResult: 1a0a9262-7858-4192-9981-b7bf0ea7483b>
```

Selenium

1. 使用 selenium 操作 cookie 时，可以获取所有 cookie 后直接使用 pickle 来保存 cookie，

```
cookies=driver.get_cookies()
pickle.dump(cookies , open(filename,"wb"))
```

注意在 load 时需要排除掉跨域的 cookie：

```
cookies = pickle.load(pkl_file)
for cookie in cookies:
    if cookie["domain"] != ".weibo.com" and cookie["domain"] !=
".passport.weibo.com" and cookie["domain"] != "current.sina.com.cn":    #排除无效
域
        driver.add_cookie(cookie)
```

2. frame 标签有 frameset、frame、iframe 三种，frameset 跟其他普通标签没有区别，不会影响到正常的定位，而 frame 与 iframe 对 selenium 定位而言是一样的，selenium 有一组方法对 frame 进行操作：

```
driver.switch_to.frame(reference)  # 切到指定 frame，可用 id 或
name(str)、index(int)、元素(WebElement)定位
driver.switch_to.parent_frame()  # 切到父级 frame，如果已是主文档，则无效果
driver.switch_to.default_content()  # 切到主文档，DOM 树最开始的<html>标签
```

(1) 切换到 frame 中

```
switch_to.frame(reference)
```

reference 是传入的参数，用来定位 frame，可以传入 id、name、index 以及 selenium 的 WebElement 对象。WebElement 也就是 find_element_by_xpath 之类的函数。

(2) 从 frame 中切回主文档

```
driver.switch_to.default_content()
```

(3) 嵌套 frame 的操作

```
driver.switch_to.frame("frame1")  # 先切到 frame1
driver.switch_to.frame("frame2")  # 再切到 frame2
```

3. 类方法里不能出现中文，因为类方法不执行该类的初始化工作，所以无法转换编码，除非在类方法里转换编码。