



# C++程序员Python 3速成

郭炜 微博 <http://weibo.com/guoweiofpku>  
<http://blog.sina.com.cn/u/3266490431>

# 程序结构

- 顺序执行，没有入口函数。

```
print ("Hello,world!")
```

- 换行就是语句的结束符号，相当于C++的 ';'

```
print ("Hello,world!")  
print ("ok")
```

# 程序结构

➤ 用缩进来替代c++中的 { }

n = 7    #变量可以直接赋值，类型不固定

if n == 5:

    print ("Hello,world!")

    print ("n=5")

else:

    print (123)

    print ("abc")

输出:

123

abc

# 程序结构

## ➤ 不能乱缩进

```
print ("Hello,world!")  
    print ("n=5")    #编译错误！ 无效缩进
```

```
if x != 13:  
    print ("13")  
    print("ok") #编译错误，不一致的缩进
```

```
if x != 13:  
    print ("13")  
    print("ok")  
print("good")    #没问题
```

# 程序结构

## ➤ 不能乱缩进

```
if x != 13:  
    print ("13")  
    print("ok")  #编译错, 意外缩进  
print("good")
```

# 程序结构

➤ 一行太长则使用 "\"和 "()"换行

```
a = 4
```

```
if a > 3 and \
```

```
    a < 9:
```

```
    print( 'ok')
```

```
if (a > 3 and
```

```
    a < 9):
```

```
    print( 'ok')
```

# 注释

- 单行注释: `"#"` 开头
- 多行注释: `'''` 开头和结尾

```
'''  
    this is comment  
    this is....  
'''
```

# 变量

- 变量不需要声明。每个变量在使用前都必须赋值。

- 变量类型可变

```
x = 5
```

```
x = "123"
```



# 数据类型

- 五个标准的数据类型:

- 数字:

- `int`        `123456899899`

- `float`    `3.2`     `1.5E6`

- `complex` `1+2j`

- 字符串                    `"hello"`

- 列表                      `[1,2,'ok',4.3]`

- 元组                      `(1,2,3)`

- 字典                      `{'a':20,'b':30,49:50}`

以上类型名相当于类名, 可用于创建对象, 如:

```
a = str()
```

# 获取数据类型

```
a = 1.5E6
```

```
b = 1+2j
```

```
print( type(a) == type(b) )           # false
```

```
print(type(a))                        # <class 'float'>
```

```
print(type(b))                        # <class 'complex'>
```

```
print(type("fdasdf"))                # <class 'str'>
```

```
print(type((1,3,3)))                  # <class 'tuple'>
```

```
print(type([1,2,3]))                  # <class 'list'>
```

```
print(type({'ok':1, '22':3}))          # <class 'dict'>
```

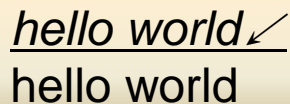
# 输入

## ➤ input()

返回输入的字符串(到回车为止, 可以带空格)

```
x = input()
```

```
print (x)
```



hello world ✓  
hello world

# 输入

➤ `input("please input:")`

先输出"please input:", 然后返回输入的字符串(到回车为止, 可以带空格),

```
x = input("please input:")  
print (x)
```

please input: hello world ✓  
hello world

# 输入

➤ `input("please input:")`

先输出"please input:", 然后返回输入的字符串(到回车为止, 可以带空格),

```
x = input("please input:")  
print (int(x)+2)
```

please input: 125✓  
127

# 输出

- print 输出一行
- print输出不换行:

```
print("this",end="") # end缺省值是换行符
print(" is ",end="")
print(8)
```

=>

```
this is 8
```

# 输出

➤ `print` 格式控制符 (和C++基本相同)

```
print ("我叫 %s 今年 %d 岁!" % ('小明', 10))
```

```
# 我叫 小明 今年 10 岁!
```

## 输入输出重定向

```
import sys
f = open("t.txt", "r")
g = open("d.txt", "w")
sys.stdin = f
sys.stdout = g
s = input()
print(s)
f.close()
g.close()
```



# 赋值语句

## ➤ 同步赋值

```
s1,s2 = 5,24  
print (s1+s2) # 29
```

## ➤ 交换变量

```
s1,s2 = 5,24  
s1,s2 = s2,s1  
print (s1) # 24  
print (s2) # 5
```

# 分支语句

```
n = 7
if n == 5:
    print("n=5")
    print("ok1")
elif n == 6:
    print("n=5")
    print("ok2")
elif n == 7:
    print("n=7")
    print("ok2")
else:
    print("else")
```

Python没有 switch语句!

# 循环语句

```
count = 0
```

```
while count < 5:
```

```
    print (count, " 小于 5")
```

```
    count = count + 1
```

```
else:
```

```
    print (count, " 大于或等于 5")
```

# 循环语句

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

```
sites = ["Baidu", "Google", "IBM", "Taobao"]    #list  
for site in sites:  
    if site == "IBM":  
        print("OK")  
        break  
    print("site: " + site)  
else:  
    print("No break")  
print("Done!")
```

```
site: Baidu  
site: Google  
OK  
Done!
```

# 循环语句

```
sites = ["Baidu", "Google", "IBM", "Taobao"]
for site in sites:
    if site == "IBM":
        print("OK")
    print("site: " + site)
else:
    print("No break")
print("Done!")
```

site: Baidu  
site: Google  
OK  
site: IBM  
site: Taobao  
No break  
Done!

# 循环语句

```
for i in range(5):    #[0,5)
    print(i)
```

```
for i in range(5,9) : #[5,9)
    print(i)
```

```
for i in range(0, 10, 3) :    #步长3
    print(i)
```

```
for i in range(-10, -100, -30) :
    print(i)
```

```
a = ['Google', 'Baidu', 'IBM', 'Taobao', 'QQ']
for i in range(len(a)):
    print(i, a[i])
```

# 循环语句

```
for letter in 'Taobao':           # 第一个实例
    if letter == 'o':             # 字母为 o 时跳过输出
        continue
    print ('当前字母 :', letter)
```

```
while True:
    pass    #什么都不做
```

# 函数

```
def Max(x,y):  
    if x > y:  
        return x  
    else:  
        return y  
print(Max(5,6)) # 6
```



# 整数类型

➤ 可以任意长，有10进制、2进制、8进制、16进制

137298372839838893838

0b101000010010100111111

0o12345675254623222233434

0x12334abcd12ABd43454

x = int()      x = int(17)

# 整数类型的运算

➤  $+$ ,  $-$ ,  $*$ ,  $\%$ , 与C++同

➤  $/$  结果是浮点数

➤  $//$  整数除法, 与C++同

➤  $x ** y$  求 $x$ 的 $y$ 次幂

➤ 位运算同C++, 但没有符号位的问题

# 布尔类型

## ➤ 常量: True, False

- True 1
- False 0
- 0 -> False 其他都是 True

```
if 2 :  
    print("ok")           # ok  
  
print(1+True )           # 2
```

## ➤ 逻辑运算符: and or not

# 布尔类型

➤ 空 str, tuple, list, dict 都是 False

```
if not ():  
    print ("ok")           # ok
```

```
if not {}:  
    print ("ok")          # ok
```

```
if not "":  
    print ("ok")          # ok
```

```
if not []:  
    print ("ok")          # ok
```

# 浮点类型

## ➤ float

```
x = float(12.334)
```

```
y = 63.3883
```

# 浮点类型

➤ 判断浮点数相等：

```
import sys

def equal_float(a,b):
    return abs(a-b) <= sys.float_info.epsilon
    #最小浮点数间隔,32位机为 2e-16

print(equal_float(1.0,1.0))
```

# 浮点类型

## ➤ sys.float\_info

```
import sys
```

```
print(sys.float_info)
```

=>

```
sys.float_info(max=1.7976931348623157e+308,  
               max_exp=1024, max_10_exp=308,  
               min=2.2250738585072014e-308, min_exp=-1021,  
               min_10_exp=-307, dig=15, mant_dig=53,  
               epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

# 浮点运算

```
import math
```

```
x = 123.50
```

```
print(round(x))    #不精确, 如print(round(119.49999999999999)) => 120
```

```
print(math.floor(x))
```

```
print(math.ceil(x))
```

```
print(12.3.is_integer())
```

```
print(12.00.is_integer())
```

```
s = str(14.25)    #转字符串
```

```
print(s)
```

```
124  
123  
124  
False  
True  
14.25
```



# 高精度浮点数decimal.Decimal

```
import decimal
```

```
from decimal import *
```

```
a = decimal.Decimal(98766)
```

```
b = decimal.Decimal("123.223232323432424244858484096781385731294")
```

```
print(b)
```

```
c = decimal.Decimal(a + b) #精度缺省为小数点后面28位
```

```
print(c)
```

```
getcontext().prec = 50 #设置精度为小数点后面50位
```

```
print(c)
```

```
c = decimal.Decimal(a + b)
```

```
print(c)
```

```
123.223232323432424244858484096781385731294
```

```
98889.22323232343242424485848
```

```
98889.22323232343242424485848
```

```
98889.223232323432424244858484096781385731294
```

# 字符串

➤ 可以用单引号、双引号、三引号括起来

```
x = "I said:'hello'"
print(x)
print('I said:"hello"')
print('''I said:'he said "hello"'. ''')
print("this \
is \
good") #字符串太长时
print("hello\nworld")
```

```
I said:'hello'
I said:"hello"
I said:'he said "hello"'.
this is good
hello
world
```

# 字符串

➤ 三双引号字符串中可以包含换行符、制表符以及其他特殊字符。

`para_str = """多行字符串可以使用制表符`

`TAB ( \t )。`

`也可以使用换行符 [ \n ]。`

`<HTML><HEAD><TITLE>`

`Friends CGI Demo</TITLE></HEAD>`

`<BODY><H3>ERROR</H3>`

`<FORM><INPUT TYPE=button VALUE=Back`

`ONCLICK="window.history.back()"></FORM>`

`</BODY></HTML>`

`"""`

`print (para_str)`

多行字符串可以使用制表符

TAB (        )。

也可以使用换行符 [

]。

<HTML><HEAD><TITLE>

Friends CGI Demo</TITLE></HEAD>

<BODY><H3>ERROR</H3>

<FORM><INPUT TYPE=button VALUE=Back  
ONCLICK="window.history.back()"></FORM>

</BODY></HTML>

# 字符串

➤ 可以用[]访问其中的子串，单个字符也是子串

```
x = "abcdefg"
```

```
print("1)" + x[0])
```

```
print("2)" + "abcdefg"[-1])  #-1表示最后一个字符
```

```
print("3)" + x[0:3])  #左闭右开的区间[0,3)
```

```
print("4)" + "abcdefg"[1:3])  #左闭右开的区间[1,3)
```

```
print("5)" + x[:5])  #默认起点为0
```

```
print("6)" + x[4:])  #默认终点为最后一个字符
```

```
print("7)" + x[:])
```

```
print("8)" + x[-3:-1])
```

```
print("9)" + x[-1:])
```

1)a  
2)g  
3)abc  
4)bc  
5)abcde  
6)efg  
7)abcdefg  
8)ef  
9)g

# 字符串

## ➤ 字符串不可修改

```
a = "1234"
```

```
print(a[2])
```

```
a[2] = "3"      # error
```

## ➤ 不转义的字符串

```
print(r'ab\ncd')      # ab\ncd
```

# 字符串

➤ 用 `in` , `not in` 判断字符串

```
a = "Hello"
```

```
b = "Python"
```

```
print("el" in a)
```

```
print("th" not in b)
```

```
print("lot" in a)
```

```
True  
False  
False
```

# 字符串函数

- count 求子串出现次数

```
s = 'thisAAbb AA'
```

```
s.count('AA')    # 2
```

- len 字符串长度

```
s = '1234'
```

```
len(s)           # 4
```

# 字符串函数

- upper, lower 转大写、小写

```
s = "abc"  
print(s.upper())    # ABC  
print(s)            # abc
```

- len 字符串长度

```
s = '1234'  
len(s)              # 4
```



# 字符串函数

- find, rfind, index, rindex 查找

找不到find 返回-1; index抛出异常

```
s="1234abc567abc12"
```

```
print(s.find("ab")) # 4
```

```
print(s.rfind("ab")) #10
```

```
try :
```

```
    s.index("afb")
```

```
except Exception as e:
```

```
    print(e) # substring not found
```

# 字符串函数

- `replace` 替换

```
s="1234abc567abc12"
```

```
b = s.replace("abc","ABC")
```

```
print(b)    # 1234ABC567ABC12
```

```
print(s)    # 1234abc567abc12
```

- `isdigit()`, `islower()`, `isupper()` 判断是否是数, 大小写等
- `startswith`, `endswith` 判断是否以某子串开头、结尾

# 字符串函数

- strip(), lstrip(), rstrip() 除去空白字符，包括空格， '\r' '\t' '\n'

```
print ( " \t12 34 \n ".strip()) # 12 34
```

```
print ( " \t12 34 5".lstrip()) # 12 34 5
```

# 字符串函数

- split 字符串分割

```
print( " \t12 34 ,ab\n ".split()) #用空格,回车,制表符分割
print( " \t12 34 ,ab,cd\n ".split(",")) #用 ' , ' 分割
y = " \t12,.34 ,ab,.cd\n ".split(",.") #用 ",." 字符串分割
print(y[1])
print('A123AA456AA7A'.split('A'))
```

分割结果是字符串列表

```
['12', '34', ',ab']
['\t12 34 ', 'ab', 'cd\n ']
34 ,ab
[' ', '123', ' ', '456', ' ', '7', ' ']
```

# 字符串

## ➤用多个字符进行分割

```
import re
a = 'Beautiful, is; better*than\nugly'
print(re.split(';| |,|\*|\n',a)) #分隔串用 | 隔开

# ['Beautiful', '', 'is', '', 'better', 'than', 'ugly']
```

# 字符串

➤ 字符串的编码在内存中的编码是unicode的。没有字符类型

```
print (ord("a"))
```

```
print(ord("好"))
```

```
print(chr(22900))
```

```
print(chr(97))
```

97

22909

奴

a

# 字符串格式化

```
x = "Hello {0} {1:10},you get ${2:0.4f}".format("Mr. ","Jack",3.2)
```

```
print(x)
```

```
x = "Hello {0} {1:>10},you get ${2:0.4f}".format("Mr. ","Jack",3.2)
```

```
print(x)
```

{序号: 宽度.精度.类型}

> : 右对齐

< : 左对齐

^ : 中齐

```
Hello Mr. Jack    ,you get $3.2000  
Hello Mr.      Jack,you get $3.2000
```

# 字符串格式化

```
print("Today is %s.%d." % ('May',21))
```

```
# Today is May.21.
```



# 字符的编码

## ➤ ASCII (ANSI) 编码

只能表示255个字符，字母、数字、标点符号等      'a' 0x61    'b' 0x62

## ➤ GB2312 (GBK) 编码

- 保留ASCII编码中的0-127的基础上，用相邻2个最高位为1的字节表示1个汉字
- 若ASCII (ANSI) 文件在记事本显示为：

abc123好的45

其十六进制内容为：

**61 62 63 31 32 33 BA C3 B5 C4 34 35**

记事本打开非文本文件可能出现半个汉字等乱码情况

# 字符的编码

## ➤ unicode编码

- 大多数操作系统在**内存**中表示字符都使用此编码。每个字符由2个字节构成。ASCII字符补第一字节为0（极个别字符用4字节）
- 几乎涵盖世界上所有的文字
- python的ord函数取字符的unicode编码  
ord('好')      22920  
ord('a')      97
- python的字符串是unicode编码

# 字符的编码

## ➤ utf8编码

- 世界上的信息以英文为主，用unicode表示比较浪费
- utf8为可变长编码(1-6字节)，常用英文字母，数字，符号为1个字节，汉字通常3个字节
- 文字传输时一般转成utf8编码，保存到文件(如网页)一般也用utf8编码
- 读取文本文件内容，要指明编码，这样操作系统读文件时会根据其编码进行识别，再自动转换为unicode编码在内存中使用

# 字符的编码

## ➤ utf8编码

- 若UTF8文件在记事本显示为：

abc123好的4

其十六进制内容为：

EF BB BF 61 62 63 31 32 33 E5 A5 BD E7 9A 84 34

前三个字节是utf8格式的标记 BOM (Byte Order Mark)

- 在记事本中显示正常的 utf8文件，在C++中用 `cin` 读取很可能不对
- 在C++中UTF-8需要用 `_w fopen(L"utf8.txt", L"r,ccs=utf-8")` 之类的办法读取

# Python字符串和字节流的互相转换

```
bs = 'ABC好的'.encode('utf-8')
print(bs)           # b'ABC\xe5\xa5\xbd\xe7\x9a\x84'
print(len(bs))      # 9
bs = 'ABC好的'.encode('gbk')
bs = bytes('ABC好的', encoding = "gbk")
print(bs)           # b'ABC\xba\xc3\xb5\xc4'
print(len(bs))      # 7
s = str(bs,encoding = "gbk") # ABC好的

print(str(b'abc\xe5\xa5\xbd\xe7\x9a\x84',encoding =
"utf-8"))           # abc好的
```

# Python源程序的编码

- pyCharm 编写的 .py 文件是 utf-8 编码

```
python test.py
```

python 缺省认为源程序是utf8编码，test.py文件应该有开头的BOM标志。如果没有，则报错：

```
SyntaxError: Non-UTF-8 code starting with '\xb7' in file  
testansi.py on line 1, but no encoding declared;
```

- 想要运行 GBK编码的源程序，需要在源程序开头加：

```
# -*- coding: GBK -*-
```

# 类型转换函数

`int(x[, base])`

将字符串x转换成整数

```
print(int("f0",16))      #240
print(int("123"))         #123
print(int("1010",2))      #10
print(int("10",8))        #8
print(int("12345678901224556677766677"))
#12345678901224556677766677
```

# 类型转换函数

<code>float(x)</code>	将字符串 <code>x</code> 转换为浮点数
<code>str(x)</code>	将对象 <code>x</code> 转换为字符串
<code>repr(x)</code>	将对象 <code>x</code> 转换为可执行字符串
<code>chr(x)</code>	将整数转换为字符
<code>ord(x)</code>	将字符转换为整数编码值(16位)



## 类型转换函数

```
print(chr(97))           # a
print(ord('好'))         # 22909
print(chr(22909))        # 好
print(str(12.334))       # 12.334
print(repr(12.334))      # 12.334
print(str("hello"))      # hello
print(repr("hello"))     # 'hello'
newstr = list('abcdafg')
print(newstr)             # ['a', 'b', 'c', 'd', 'a', 'f', 'g']
newstr[4] = 'e'
str = ''.join(newstr)
print(str)               # abcdefg
```

# 用is 判断是否是同一个对象

```
a = (1,2,3)
b = (1,2,3)
print( a == b )      # true
print(a is b)        # false

a = [1,2,3]
b = [1,2,3]
print(a == b)        # true
print(a is b)        # false

print([1,2,3] is [1,2,3])  # false
print((1,2,3) is (1,2,3))  # false
```

# 用is 判断是否是同一个对象

```
a = 1.2
b = 1.2
print( a is b )          # true

a = "this"
b = "this"
print( a is b )          # true
print( a is "this" )     # true

a = 12345678901
b = 12345678901
print(a is b )           # true
```

# 元组

- 一个元组由数个逗号分隔的值组成, 前后可加括号
- 元组相当于C++中的 `vector`, 各种操作复杂度亦然。可认为是个指针数组, 数组元素不可修改, 不可增删元素, 元素指向的东西可以修改

```
t = 12345, 54321, 'hello!'
```

```
print(t[0])           # 12345
```

```
print(t)              # (12345, 54321, 'hello!')
```

```
u = t, (1, 2, 3, 4, 5)
```

```
print(u)              # ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

```
t[0] = 88888          #元组的值不能修改, 运行错误
```

# 元组

# 元组可以包含可修改的对象

```
v = ("hello", [1, 2, 3], [3, 2, 1]) # [1, 2, 3]是列表
```

```
v[1][0] = 'world'
```

```
print(v)          # ('hello', ['world', 2, 3], [3, 2, 1])
```

```
print(len(v))     # 3 求长度
```

# 元组

## ➤ 单元素的元组

```
empty = ()    #空元组
```

```
singleton = 'hello',    # <-- note trailing comma
```

```
print(len(empty))
```

```
print(len(singleton))
```

```
x = ('hello',)    #无逗号则x为字符串
```

```
print(x)
```

```
0
1
('hello',)
```

# 元组

## ➤ 用下标访问元组

```
tup1 = ('Google', 'Runoob', 1997, 2000)
```

```
tup2 = (1, 2, 3, 4, 5, 6, 7 )
```

```
print ("tup1[0]: ", tup1[0])
```

```
print ("tup2[1:5]: ", tup2[1:5])
```

```
tup1[0]: Google  
tup2[1:5]: (2, 3, 4, 5)
```

# 元组

➤ 元组中的元素值是不允许修改的，但可以对元组进行连接组合

```
tup1 = (12, 34.56);
```

```
tup2 = ('abc', 'xyz')
```

# 以下修改元组元素操作是非法的。

```
# tup1[0] = 100
```

# 创建一个新的元组

```
tup3 = tup1 + tup2;
```

```
print (tup3)
```

```
(12, 34.56, 'abc', 'xyz')
```



# 元组

## ➤ 元组运算和迭代

```
x = (1,2,3) * 3
```

```
print(x)
```

```
print( 3 in (1,2,3))
```

```
for i in (1,2,3):
```

```
    print(i)
```

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

```
True
```

```
1
```

```
2
```

```
3
```

# 元组拷贝

```
x = (1,2,3)
```

```
b = x
```

```
print(b is x) # true
```

```
x += (100,)
```

```
print (x)      # (1, 2, 3, 100)
```

```
print (b)      # (1, 2, 3)
```

# 列表

➤ 列表相当于C++中的 `vector`, 各种操作复杂度亦然。可认为是个指针数组

```
empty = []    #空表
```

```
list1 = ['Google', 'Runoob', 1997, 2000];
```

```
list2 = [1, 2, 3, 4, 5, 6, 7 ];
```

```
print ("list1[0]: ", list1[0])
```

```
print ("list2[1:5]: ", list2[1:5])
```

```
list1[2] = 2001
```

```
print ("更新后的第三个元素为  :", list1[2])
```

```
list1[0]: Google
```

```
list2[1:5]: [2, 3, 4, 5]
```

```
更新后的第三个元素为：2001
```

# 列表

```
del list1[2]

print ("删除第三个元素 : ", list1)

list1 += [100]

print(list1)

a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n] #a若变, x也变

print( x)

print(x[0])

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

删除第三个元素 : ['Google', 'Runoob', 2000]  
['Google', 'Runoob', 2000, 100]  
[['a', 'b', 'c'], [1, 2, 3]]  
['a', 'b', 'c']  
b

# 列表的切片

➤ 列表的切片返回新的列表

```
a = [1,2,3,4]
```

```
b = a[1:3]
```

```
print(b)          # [2, 3]
```

```
b[0] = 100
```

```
print(b)          # [100, 3]
```

```
print(a)          # [1, 2, 3, 4]
```

# 列表拷贝

```
a = [1,2,3,4]
```

```
b = a    #a,b是同一个对象
```

```
print(b is a)    # True
```

```
b[0] = 5
```

```
print(a)          # [5, 2, 3, 4]
```

```
b += [10]          # [5, 2, 3, 4, 10] ,原地添加, 和元组不同
```

```
print(a)          # [5, 2, 3, 4, 10]
```

```
print(b)          # [5, 2, 3, 4, 10]
```

# 列表拷贝

```
a = [1,2,3,4]
```

```
b = a[:]    #b是a的拷贝
```

```
b[0] = 5
```

```
print(a)      # [1, 2, 3, 4]
```

```
b += [10]
```

```
print(a)      # [1, 2, 3, 4]
```

```
print(b)      # [5, 2, 3, 4, 10]
```

# 列表深拷贝

```
a=[1,[2]]
```

```
b=a[:]
```

```
b.append(4)
```

```
print(b) #=> [1, [2], 4]
```

```
a[1].append(3)
```

```
print(a) #=> [1, [2, 3]]
```

```
print(b) #=> [1, [2, 3], 4]
```

未能进行深拷贝！



# 列表深拷贝

```
import copy
```

```
a = [1, [2]]
```

```
b = copy.deepcopy(a)
```

```
b.append(4) #=>[1, [2], 4]
```

```
print(b)
```

```
a[1].append(3)
```

```
print(a) #=>[1, [2, 3]]
```

```
print(b) #=>[1, [2], 4]
```

# 列表生成式

```
[x * x for x in range(1, 11)]
```

```
⇒ [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
[x * x for x in range(1, 11) if x % 2 == 0]
```

```
⇒ [4, 16, 36, 64, 100]
```

```
[m + n for m in 'ABC' for n in 'XYZ']
```

```
⇒ ['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

```
L = ['Hello', 'World', 18, 'Apple', None]
```

```
[s.lower() for s in L if isinstance(s, str)]
```

```
⇒ ['hello', 'world', 'apple']
```

```
[s for s in L if isinstance(s, int)] ⇒ [18]
```

# 处理输入

```
try:
    while True:
        s = input() #读取一行
        if s.strip() == "":
            continue;
        a = s.split(' ')
        b = [x for x in a if x != ''] #去除空串
        name, age, gpa = b[0], b[1], b[2]
        print(name, age, gpa)
except EOFError:
    pass
```

标准输入:

jack 12 3.5  
marry 34 7.8  
Jane 34 7.8

输出:

jack 12 3.5  
marry 34 7.8  
Jane 34 7.8

# 列表相关函数

- `append (value)` 添加1个元素
- `extend (list)` 添加其他表中的元素
- `insert,remove,reverse`
- `index` 查找
- `len(x)` 求列表x长度

# 列表相关函数

```
a = [1,2,3]
```

```
b = [5,6]
```

```
a.append(b)           # [1, 2, 3, [5, 6]]
```

```
a.extend(b)           # [1, 2, 3, [5, 6], 5, 6]
```

```
a.insert(1,'K')        # [1, 'K', 2, 3, [5, 6], 5, 6]
```

```
a.insert(3,'K')        # [1, 'K', 2, 'K', 3, [5, 6], 5, 6]
```

```
a.remove('K')          # [1, 2, 'K', 3, [5, 6], 5, 6]
```

```
a.reverse()           # [6, 5, [5, 6], 3, 'K', 2, 1]
```

```
a.index('K')           # 4
```

```
print(len(a))          # 7
```

# 列表相关函数

```
print('m' in a)                # False

try:
    print(a.index('m'))
except Exception as e:
    print(e)                    # 'm' is not in list
```

# 列表相关函数

- `map(function, sequence)`, 将一个列表(元组)映射到另一个列表(元组)
- `filter(function, sequence)`, 按照所定义的函数过滤掉列表(元组)中的一些元素

```
def f1(x):
```

```
    return x * 2
```

```
def f2(x):
```

```
    return x % 2 == 0
```

```
ls = [1,2,3,4,5]
```

```
ls1 = list(map(f1,ls))      #map延时求值
```

```
print(ls1)  #=>[2, 4, 6, 8, 10]
```

```
ls1 = list(filter(f2,ls))  #filter 延时求值
```

```
print(ls1)  #=>[2, 4]
```

# 列表相关函数

```
def f1(x):
```

```
    return x * 2
```

```
def f2(x):
```

```
    return x % 2 == 0
```

```
ls = (1,2,3,4,5)
```

```
ls1 = tuple(map(f1,ls))
```

```
print(ls1) #=>(2, 4, 6, 8, 10)
```

```
ls1 = list(filter(f2,ls))
```

```
print(ls1) #=> [2, 4]
```



# 列表相关函数

## ➤ map 高级用法

```
def abc(a, b, c):  
    return a*10000 + b*100 + c  
  
list1 = [11,22,33]  
list2 = [44,55,66]  
list3 = [77,88,99]  
  
x = list(map(abc,list1,list2,list3))  
print(x) #=> [114477, 225588, 336699]
```

# 列表相关函数

- `reduce(function, sequence, startValue)`, 将一个列表累积(accumulate)起来

```
from functools import reduce
```

```
def f2(x,y):
```

```
    return x+y
```

```
ls = [1,2,3,4,5]
```

```
print(reduce(f2,ls)) #=>15
```

```
print(reduce(f2,ls,10)) #=>25
```

# 列表相关函数

➤ reduce的实现:

```
def myReduce(function, sequence, startValue):  
    for x in sequence:  
        startValue = function(startValue, x)  
    return startValue
```

## 用列表定义二维数组

```
array = [1, 2, 3]
```

```
matrix = [array*3]
```

```
print (matrix) # [[1, 2, 3, 1, 2, 3, 1, 2, 3]] ,非二维数组
```

```
array = [0, 0, 0]
```

```
matrix = [array] * 3
```

```
print(matrix) # [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

```
matrix[0][1] = 1
```

```
print(matrix) # [[0, 1, 0], [0, 1, 0], [0, 1, 0]] ,非二维数组
```

```
#matrix[0],matrix[1],matrix[2]指向相同的一维数组array
```

# 用列表定义二维数组

## ➤ 正确做法:

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

```
print(matrix)    # [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
print(matrix[1][1])    # 5
```

```
matrix = [[0 for i in range(3)] for i in range(3)]
```

```
print(matrix)    # [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

```
matrix = [[i*3+j for j in range(3)] for i in range(3)]
```

```
print(matrix)    # [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

## 用元组定义二维数组

```
matrix = ((1, 2, 3), (4, 5, 6), (7, 8, 9))  
print(matrix) # ((1, 2, 3), (4, 5, 6), (7, 8, 9))  
print(matrix[1][1]) # 5
```

# 元组和列表互转

```
a=[1,2,3]
```

```
b=tuple(a)      # (1,2,3)
```

```
c=list(b)       # [1,2,3]
```

```
t = (1, 3, 2)
```

```
(a, b, c) = t    # a = 1, b = 3, c = 2
```

```
s = [1,2,3]
```

```
[a,b,c] = s      # a = 1, b = 2, c = 3
```

# 列表的排序

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

```
a.sort() # [1, 2, 3, 4, 5, 6, 7]
```

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

```
b = sorted(a) # b: [1, 2, 3, 4, 5, 6, 7], a不变
```

```
a = [25,7,16,33,4,1,2]
```

```
a.sort(reverse = True) # [25, 7, 16, 33, 4, 1, 2] sorted类似
```



# 列表的排序

## ➤ 自定义比较函数 key

```
def myKey(x):    # 函数
    return x % 10
```

```
a = [25,7,16,33,4,1,2]
```

```
a.sort(key = myKey)    # key是函数，sort按对每个元素调用该函数的返回值从
                        小到大排序
```

```
# [1, 2, 33, 4, 25, 16, 7]  按个位数排序
```

```
sorted("This is a test string from Andrew".split(),
       key=str.lower))
```

```
# ['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

不区分大小写排序

# 列表的排序

## ➤ 用不同关键字排序以及多级排序

```
from operator import *
```

```
students = [  
    ('John', 'A', 15), # 姓名, 成绩, 年龄  
    ('Mike', 'B', 12),  
    ('Mike', 'C', 18),  
    ('Bom', 'D', 10)]
```

```
students.sort(key = lambda x: x [2] ) # 按年龄排序
```

```
# [('Bom', 'D', 10), ('Mike', 'B', 12), ('John', 'A', 15), ('Mike', 'C', 18)]
```

```
students.sort(key = lambda x: x [0] ) # 按姓名排序
```

```
# [('Bom', 'D', 10), ('John', 'A', 15), ('Mike', 'B', 12), ('Mike', 'C', 18)]
```

```
sorted(students, key= itemgetter(0,1)) # 先按姓名后按成绩排序
```

```
# [('Bom', 'D', 10), ('John', 'A', 15), ('Mike', 'B', 12), ('Mike', 'C', 18)]
```

# 元组的排序

➤ 元组不能修改，因此无`sort`，可以用`sorted`得到新的排序后的元组

```
from operator import *
```

```
students = (  
    ('John', 'A', 15), # 姓名, 成绩, 年龄  
    ('Mike', 'B', 12),  
    ('Mike', 'C', 18),  
    ('Bom', 'D', 10))
```

```
print(sorted(students, key = itemgetter(0,1)))
```

```
# [('Bom', 'D', 10), ('John', 'A', 15), ('Mike', 'B', 12), ('Mike', 'C', 18)]
```

# 字典

➤ 键唯一且不可变(字符串、整数、元组、对象....., 不可是list)

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

```
dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}  
print ("dict['Name']: ", dict['Name'])    # dict['Name']: Runoob  
print ("dict['Age']: ", dict['Age'])       # dict['Age']: 7
```

```
dict['Age'] = 8;          # 更新 Age  
dict['School'] = "Pku"   # 添加
```

# 字典

```
print(dict['School'])          # Pku

del dict['Class']

print(dict)    # {'School': 'Pku', 'Age': 8, 'Name': 'Runoob'}
```

```
scope={}          #空字典

scope['a'] = 3      # 加元素

scope['b'] = 4

print(scope['a'])   # 3

print(scope['c'])   # KeyError: 'c' 键不存在，产生异常
```

# 字典的构造

```
>>> items = [('name', 'Gumby'), ('age', 42)]
```

```
>>> d = dict(items)
```

```
>>> d
```

```
{'name': 'Gumby', 'age': 42}
```

```
>>>d=dict(name='Gumby',age=42)
```

```
>>> d
```

```
{'name': 'Gumby', 'age': 42}
```

```
>>> 'age' in d
```

```
True
```

# 字典的赋值

字典变量本质上是指针

```
d={'name': 'Gumby', 'age': 42}
b = d;
b['name'] = "Tom"
print(d)      # {'name': 'Tom', 'age': 42}
```

# 字典

```
>>> phonebook = {'Alice' : '2341', 'Beth' : '9102', 'Cecil'  
: '3258'}  
>>> "Cecil's phone number is %(Cecil)s." % phonebook  
>>> "Cecil's phone number is 3258."
```

对比:

```
print ("我叫 %s 今年 %d 岁!" % ('小明', 10))
```



# 字典的方法

## ➤ clear

```
d={'name': 'Gumby', 'age': 42}
d.clear()    # {}
```

## ➤ keys

```
d={'name': 'Gumby', 'age': 42, 'GPA': 3.5}
if 'age' in d.keys():
    print(d['age'])           # 42
for x in d.keys():
    print(x, end=", ")       # name, GPA, age,
```

# 字典的方法

➤ items

➤ values

```
d={'name': 'Gumby', 'age': 42, 'GPA': 3.5}
for x in d.items():
    print(x,end=",")
# ('name', 'Gumby'), ('age', 42), ('GPA', 3.5),
for x in d.values():
    print(x,end=",")
# Gumby,42,3.5,
```

# 字典的方法

## ➤ copy 浅拷贝

```
x = {'username': 'admin', 'machines': ['foo', 'bar', 'baz']}  
y = x.copy()  
y['username'] = 'mlh'  
y['machines'].remove('bar')  
print(y)  
print(x)  
=>
```

```
{'machines': ['foo', 'baz'], 'username': 'mlh'}  
{'machines': ['foo', 'baz'], 'username': 'admin'}
```

# 字典的深拷贝

```
import copy
x = {'username': 'admin', 'machines': ['foo', 'bar', 'baz']}
y = copy.deepcopy(x)
y['username'] = 'mlh'
y['machines'].remove('bar')
print(y)
print(x)
=>
{'username': 'mlh', 'machines': ['foo', 'baz']}
{'username': 'admin', 'machines': ['foo', 'bar', 'baz']}
```

# 遍历字典

## ➤ items

```
x = {'username': 'admin', 'machines': ['foo', 'bar', 'baz'],  
     'Age': 15}
```

```
for i in x.items():  
    print(i[0])  
    print(i[1])
```

=>

**`machines`**

**`['foo', 'bar', 'baz']`**

**`Age`**

**`15`**

**`username`**

**`admin`**

# 字典的键

```
a = (1,2,3)
b = (1,2,3)    # a is b 为 False
d = {a:60,b:70,(1,2,3):80,(1,2,3):50}
print(d[a])    # 80
print(d[b])    # 80
print(d[(1,2,3)])    # 80
for x in d.keys():
    print(x)
a = "this"
b = "this"    # a is b 为 True
d = {a:60,b:70,"this":90}
print(d[a])    # 90
print(d[b])    # 90
```

# 字典的键

```
a = (1,2,3)
b = (1,2,3)    # a is b 为 False
d = {a:60,b:70,(1,2,3):80, (1,2,3):50}
print(d[a])    # 80
print(d[b])    # 80
print(d[(1,2,3)])    # 80
for x in d.keys():
    print(x)      # (1, 2, 3)
```

# 字典的键

```
a = "this"
b = "this"    # a is b 为 True
d = {a:60,b:70,"this":90}
print(d[a])    # 90
print(d[b])    # 90
print(d["this"])    # 90
for x in d.keys():
    print(x)      # this
```



# set

➤ 相当于c++的 `unordered_set`, 哈希表, 元素不可重复, 元素必须可哈希

`s = {1, 2, 3}`                      `# s是一个 set`

- `s.add( x )` 将元素 `x` 添加到集合`s`中, 若重复则不进行任何操作
- `s.update( x )` 将集合 `x` 并入原集合`s`中, `x` 还可以是列表, 元组, 字典等, `x` 可以有多个, 用逗号分开
- `s.discard( x )` 将 `x` 从集合`s`中移除, 若`x`不存在, 不会引发异常
- `s.remove( x )` 将 `x` 从集合`s`中移除, 若`x`不存在, 会引发异常
- `s.clear()` 清空
- `s.pop()`                      随机删除并返回集合`s`中某个值
- `x in s`                      判断`x`是否在`s`里面
- `s.union( x )` 返回`s`与集合`x`的并集, 不改变原集合`s`, `x` 也可以是列表, 元组, 字典。
- `s.intersection( x )` 返回`s`与集合`x`的并集, 不改变`s`, `x` 也可以是列表, 元组, 字典。

# set

- `s.difference( x )` 返回在集合s中而不在集合 x 中的元素的集合，不改变集合s， x 也可以是列表，元组，字典。
- `s.symmetric_difference( x )` 返回s和集合x的对称差集，即只在其中一个集合中出现的元素，不改变集合s， x 也可以是列表，元组，字典。
- `s.issubset( x )` 判断 集合s 是否是 集合x 子集
- `s.issuperset( x )` 判断集合x 是否是集合s的子集

# 函数

```
def Max(x,y):  
    if x > y:  
        return x  
    else:  
        return y  
print(Max(5,6))  
=> 6
```

# 函数

➤函数参数都是传值

➤Python对象都是指针，类似于java

```
def func(x):  
    x = "bbb"  
y = "a"  
func(y)  
print(y)      #a , 字符串不是指针
```

```
def modify(x):  
    x[2] = "OK"
```

```
y = [1,2,3]  
modify(y)  
print (y)    # [1, 2, 'OK']
```

# 函数参数传递

►参数都是传值

```
class A:
    n = 0
class B:
    n = 100

def Swap1(x,y):
    tmp = x.n
    x.n = y.n
    y.n = tmp
def Swap2(x,y):
    x,y=y,x
```

```
a = A()
a.n = 100
b = A()
b.n = 200
Swap1(a,b)
print(a.n,b.n) # 200 100
a = 6
b = 7
Swap2(a,b)
print(a,b)     # 6 7
```

# 函数参数传递

## ➤ 默认参数和实参带名字

```
def func(a, b=1, c=2):  
    print ('a=', a , 'b=', b, 'c=', c)
```

```
func(10,20)           # a= 10 b= 20 c= 2
```

```
func(30, c=40)        # a= 30 b= 1 c= 40
```

```
func(c=50, a=60)      # a= 60 b= 1 c= 50
```

# 函数参数传递

## ► 参数个数不定的函数

```
def func(a, *b):  
    print(b)  
c = [1,2,3]  
func(1,2,'ok',c)
```

#b是个元组

# (2, 'ok', [1, 2, 3])

```
def func(**b):  
    print(b)
```

#b是个字典

```
func(p1=2,p2=3,p3=4) #{'p2': 3, 'p3': 4, 'p1': 2}
```

# 函数中使用全局变量

➤ 函数中用到的变量，不加声明则都是局部变量

```
x = 100
def func():
    x = 10
    print('in func,x is', x)
func()
print('global x is', x)
```

# in func,x is 10  
# global x is 100



# 函数中使用全局变量

## ► 在函数中使用全局变量

```
x = 100
def func():
    global x
    global y
    print(x,y)          # 若无global声明, 则x,y都没定义, 出错
    x = 10
    print('change x into', x)
y = 1000
func()
# 100 1000
# change x into 10
print('global x is', x)    # global x is 10
```

# 多文件编程

➤ t.py


```
def hello():  
    print('hello from t')  
haha = "ok"
```

➤ a.py

```
from t import hello,haha  
#或者: from t import *  
hello()  
print('haha=',haha)
```

```
# hello from t  
# haha= ok
```

# 用pack和unpack把数据处理成二进制流

```
import struct
a = 20
b = 400
str = struct.pack("ii", a, b)
print ('length: ', len(str))           # length: 8
print (str)                             # 
b'\x14\x00\x00\x00\x90\x01\x00\x00'
print (repr(str))   # b'\x14\x00\x00\x00\x90\x01\x00\x00'

tp = struct.unpack("ii", str)  #unpack的结果是元组
print( 'length: ', len(tp))   # length: 2
print (tp)                     # (20, 400)
print (repr(tp))               # (20, 400)
print(100+tp[1])               # 500
```

# 面向对象

# object 类

- 所有类自动派生自object 类
- object类有以下成员函数：

<code>__init__</code>	构造函数
<code>__eq__</code>	<code>==</code>
<code>__lt__</code>	<code>&lt;</code>
<code>__le__</code>	<code>&lt;=</code>
<code>__ge__</code>	<code>&gt;=</code>
<code>__gt__</code>	<code>&gt;</code>
<code>__ne__</code>	<code>!=</code>
<code>__str__</code>	强制转换成str
<code>__repr__</code>	强制转换成python可执行字符串
<code>__del__</code>	析构函数
<code>.....</code>	

# 类的写法

➤ 构造函数和析构函数都只能有一个

```
class A:
```

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y  
    def func(self):  
        self.xx = 9  
    def __del__(self):  
        print(self.x , "destroyed")
```

```
a = A(1,3)
```

```
b = A(2,3)
```

```
a = 8    # 1 destroyed
```

# 类的写法

➤ 对象的成员变量可以随时添加

```
class A:
```

```
    def __init__(self,x,y=0):  
        self.x = x  
        self.y = y  
    def func(self):  
        self.xx = 9
```

```
a = A(1,3)
```

```
b = A(2,3)
```

```
a.n2 = 28
```

```
print(a.n2) # 28
```

```
a.func()
```

```
print(a.xx) # 9
```

```
print(b.n2) # error
```

```
print(b.xx) # error
```

# 类的写法

```
import math
class Point:
    def __init__(self,x=0,y=0): #构造函数,成员变量可以在此初始化
        self.x = x
        self.y = y
    def distance_from_origin(self):
        return math.hypot(self.x,self.y) #求点到 (0,0) 距离
    def __eq__(self,other): #相当于重载了 == 重新实现这个,就不是
                            #可hash,就不能放到字典
        return self.x == other.x and self.y == other.y
    def __lt__(self,other) : # less than 相当于重载 <
        if self.x != other.x :
            return self.x < other.x
        else :
            return self.y < other.y
```



# 类的写法

```
def __str__(self): #相当于重载类型转换构造函数 str
    return "({0.x},{0.y})".format(self)
```

```
def __repr__(self):
    #类似于 __str__, 返回值是个 python可执行字符串
    return "Point({0.x},{0.y})".format(self)
```

# 类的写法

```
a = Point(3,5)
b = Point(3,5)
c = Point(3,6)
```

```
print(b.distance_from_origin()) #=> 5.8309518948453
print(b == a) # => True 若无 __eq__ 则为 false
print(a < c) # => True 若无 __lt__ 则无定义, 错
print(a is b) # => False, 比对象地址
print(isinstance(a, Point)) # => True
p = eval(repr(a))
print(p == a) # => true
print(p is a) # => False
print(p) #=> (3,5)
```

# 类的写法

## ➤ 静态成员变量

```
class A:
    n = 100 #静态成员变量
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def print1(self):
        print(a.n,self.n)

print (A.n)    #=> 100
b = A(3,2)
b.print1()      #=> NameError: name 'a' is not defined
a = A(1,2)
b.print1()      #=> 100 100
a.n = 20        #a对象增加非静态成员变量 n
print(A.n,a.n)  #=> 100 20
a.print1()      #=> 20 20
b.print1()      #=> 20 100
print(b.n)      #=> 100
```

# 类的写法

## ➤ 静态方法和类方法

```
class A(object):  
    n = 10  
    def __init__(self,x):  
        self.x = x  
  
    @staticmethod    #静态方法  
    def func1():  
        print("in func1",A.n)  
        A.n = 20  
  
    @classmethod    #类方法  
    def func2(cls,x):  
        print("in func2",cls.n,x)  
        print(str(cls))  
  
A.func1()    #=>10  
A.func2(100)    #=>in func2 20 100  
               <class '__main__.A'>
```

# 类的写法

## ➤私有属性

```
class A:
```

```
    __p = 20 #私有类属性, 以 __ 开头
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        self.__q = 30 #私有实例属性
```

```
a = A()
```

```
print(a.__q) #error , 没有 __q成员变量
```

```
print(a._A__q) #=>30 实际上__q改名为 _A__q
```

```
print(a._A__p) #=>20
```

```
a._A__p = 39 #新加一个成员变量
```

```
print(A._A__p) #=>20
```

```
print(a._A__p) #=>39
```

# 类的写法

## ➤使用property（方法一）

```
class foo:
    def __init__(self):
        self.name = 'yoda'
        self.work = 'master'
    def get_person(self):
        return self.name, self.work
    def set_person(self, value):
        self.name, self.work = value
    person = property(get_person, set_person)
```

```
A3 = foo()
print(A3.person) #=>('yoda', 'master')
A3.person = 'skylaer', 'programer'
print(A3.person) #=>('skylaer', 'programer')
```

# 类的写法

## ➤使用property（方法二）

```
class foo:
```

```
    def __init__(self):  
        self.name = 'yoda'  
        self.work = 'master'
```

```
@property
```

```
def person(self):  
    return self.name, self.work
```

```
@person.setter    # 如果不指定setter属性，那么无法从类的外部对它  
                  # 的值进行设置， 这对于只读特性非常有用
```

```
def person(self, value):  
    self.name, self.work = value
```

```
A3 = foo()
```

```
print(A3.person) #=>('yoda', 'master')
```

```
A3.person = 'skylaer', 'programer'
```

```
print(A3.person) #=>('skylaer', 'programer')
```

# 可哈希(hashable)

- 不可变的内置对象，如元组，字符串，是可哈希的
- 字典，列表，集合set是不可哈希的
- 自定义对象默认可哈希，哈希值为其对象id。
- 重载了 `__eq__` 后不可哈希，再重载 `__hash__` 则又为可哈希



# 对象作为字典的Key

➤对象作为字典的关键字时，实际上是以对象的地址，而非内容作为关键字

```
class P:
```

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
print(P(1,2) is P(1,2))    # false
```

```
a = P(1,1)
```

```
b = P(1,1)
```

```
d = {a:10,b:"ok",P(1,1):1.2}
```

```
print(d[a])                # 10
```

```
print(d[b])                # ok
```

```
print(d[P(1,1)])           # KeyError: <__main__.P object at 0x028905D0>
```

# 对象的值作为字典的Key

➤ 重写 `__eq__` 和 `__hash__`

```
import re

class A:
    def __init__(self,x):
        self.x = x
    def __eq__(self,other):
        return self.x == other.x
    def __hash__(self):
        return 0
```

```
a = A(3)
b = A(3)
d = {A(5):10,A(3):20}
print(d[a])    # 20
```

# 运算符重载

➤+, +=, -, -=, \* / \*\* ^ << | & 都能以类似的形式重载

```
class A:
```

```
    def __init__(self,x):
```

```
        self.x = x
```

```
    def __iadd__(self,other):    # +=
```

```
        return self.x + other    # return A(self.x + other) 更合适
```

```
    def __add__(self,other):    # + 对象在前
```

```
        return self.x + other
```

```
    def __radd__(self,other):    # + 对象在后
```

```
        return self.x + other
```

```
    def __sub__(self,other):
```

```
        return self.x - other
```

```
a = A(10)
```

```
print(a+10)    #=> 20
```

```
print(a-5)     #=>5
```

```
print(100+a)    #=>110
```

```
a += 10    # += 的返回值赋值给a
```

```
print(a.x)    # error
```

```
print(a)      #=>20
```

# 运算符重载

## ➤ 索引重载

```
class indexer:  
    def __getitem__(self, index): #iter override  
        return index ** 2
```

```
X = indexer()  
for i in range(5):  
    print (X[i])
```

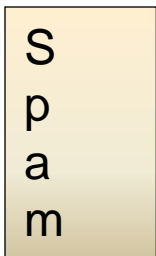
0
1
4
9
16

# 运算符重载

## ➤ 迭代重载

```
class stepper:
    def __getitem__(self, i):
        return self.data[i]

X = stepper()
X.data = 'Spam'
for item in X: #call __getitem__
    print (item)
```



S  
p  
a  
m

# 运算符重载

## ➤输出重载

```
class A:  
    n = 0  
    def __str__(self):  
        return str(self.n)
```

```
a = A()  
a.n = 12  
print(a)
```

12

# 运算符重载

## ➤Call调用函数重载

```
class Prod:
```

```
    def __init__(self, value):  
        self.value = value
```

```
    def __call__(self, other):  
        return self.value * other
```

#相当于C++ operator()

```
p = Prod(2) #call __init__  
print (p(1)) #call __call__  
print (p(2))
```

2

4

# 对象列表的排序

➤ 重载 `__lt__` 后对象列表可以排序

```
class Point:
    def __init__(self,x=0,y=0):
        self.x = x
        self.y = y
    def __lt__(self,other) : # less than 相当于重载 <
        if self.x != other.x :
            return self.x < other.x
        else :
            return self.y < other.y
    def __str__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"
a = [Point(3,5),Point(2,1),Point(4,6),Point(9,0)]
a.sort()
for x in a:
    print(x,end="") # (2,1)(3,5)(4,6)(9,0)
```



# 对象列表的排序

➤ 不重载 `__lt__` 对象列表也可以排序

```
from operator import *
```

```
class Point:
```

```
    def __init__(self,x=0,y=0):
```

```
        self.x = x
```

```
        self.y = y
```

```
    def __str__(self):
```

```
        return "(" + str(self.x) + "," + str(self.y) + ")"
```

```
a = [Point(3,5),Point(2,1),Point(4,6),Point(9,0)]
```

```
a.sort(key=attrgetter('x','y')) #先按x后按y排序
```

```
for x in a:
```

```
    print(x,end=" ")
```

```
print("")
```

```
a.sort(key=lambda p: p.y) #按y排序
```

```
for x in a:
```

```
    print(x,end=" ")
```

# 泛型

- 变量类型本来就是可变的，因此任何函数都是相当于模板

```
class A:
```

```
    n = 0
```

```
class B:
```

```
    n = 100
```

```
def Swap(x,y):
```

```
    tmp = x.n
```

```
    x.n = y.n
```

```
    y.n = tmp
```

```
a = A()
```

```
a.n = 100
```

```
b = B()
```

```
b.n = 200
```

```
Swap(a,b)
```

```
print(a.n,b.n)
```

200 100

# 继承和多态

► 本来对象类型就是运行时确定，没有明显的多态

```
class A:
```

```
    def __init__(self,x,y):
```

```
        self.x = x
```

```
        self.y = y
```

```
    def func(self):
```

```
        print("A::func",self.x,self.y)
```

```
class B(A): #派生类
```

```
    def __init__(self,x,y,z):
```

```
        A.__init__(self,x,y) #调用基类构造函数
```

```
        self.z = z
```

```
    def func(self):
```

```
        A.func(self)
```

```
        print("B::func",self.x,self.y,self.z)
```

```
a = A(1,2)
```

```
b = B(1,2,3)
```

```
a.func()
```

```
b.func()
```

=>

```
A::func 1 2
```

```
A::func 1 2
```

```
B::func 1 2 3
```

# isinstance用法

```
>>> isinstance(5,int)
True
>>> isinstance('ab',str)
True
>>> isinstance(5,float)
False
>>> isinstance(5.0,float)
True
>>> isinstance([4,5],list)
True
>>> isinstance((4,5),tuple)
True
```

# isinstance用法

```
class A:  
    pass  
class B(A):  
    pass  
a = A()  
b = B()
```

```
print(isinstance(a,A))    #true  
print(isinstance(b,A))    #true  
print(isinstance(a,B))    #false  
print(isinstance(b,B))    #true
```

# 函数式程序设计

# 函数式程序设计

➤ 函数可以用来给变量赋值

```
def Min(x,y):  
    if x < y:  
        return x  
    else:  
        return y
```

```
f = abs  
print(f(-3))    #=> 3  
f = Min  
print(f(30,4))  #=>4
```

# 函数式程序设计

## ➤ 函数可以作为函数的参数

```
class A:
    def __init__(self,n):
        self.n = n
    def __call__(self,x):
        return self.n + x
```

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

```
print(add(1,10,abs)) #=> 11
print(add(1,10,A(5))) #=> 21
```



# 函数式程序设计

## ➤ lambda表达式

```
f = lambda x,y,z:x+y+z
print(f(1,2,3)) # 6
(lambda x,y,z:x+y+z)(1,2,3) # 6
```

```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
low = 3
high = 7
x = list(filter(lambda x, l=low, h=high: h>x>l, lst))
print(x) # [4, 5, 6]
```

# 函数式程序设计

➤ lambda表达式可以作为函数的返回值和参数

```
def Inc(x):  
    return x + 1  
def Square(x):  
    return x * x  
def combine(f,g):  
    return lambda x:f(g(x))  
  
print(combine(Inc,Square)(4)) #=>17
```

```
b = lambda x:lambda y:x+y  #b是lambda表达式, 该表达式返回值是lambda表达式  
a = b(3)  
print(a(2)) #=> 5
```

# 函数式程序设计

➤ 偏应用函数 (Partial Application) 可以固定函数的某些参数:

```
from functools import partial
```

```
add = lambda a, b: a + b
```

```
add1024 = partial(add, 1024)  # a固定是1024
```

```
print(add1024(1))  #=> 1025
```

```
print(add1024(10))  #=> 1034
```

# 迭代器

- 可以用 `for i in x:` 形式遍历的对象 `x`, 称为可迭代对象
- 可迭代对象必须实现迭代器协议, 即实现 `__iter__()` 和 `__next__()` 方法
- `__iter__()` 方法返回对象本身
- `__next__()` 方法返回下一个元素

# 迭代器

```
x = [1,2,3,4]
```

```
it = iter(x) #获取x上的迭代器
```

```
print(next(it))
```

```
print(next(it))
```

```
print(next(it))
```

```
print(next(it))
```

```
print(next(it))
```

迭代器在容器末尾时，若再执行`next`，  
则会抛出异常

```
1  
2  
3  
4
```

```
Traceback (most recent call last):  
  File "D:/studypython/t.py", line 7, in  
<module>  
    print(next(it))  
StopIteration
```

# 迭代器

```
x = [1,2,3,4]
```

```
it = iter(x)
```

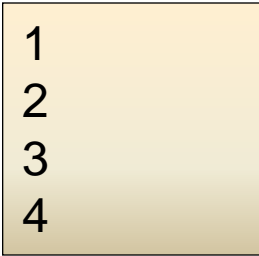
```
while True:
```

```
    try:
```

```
        print(next(it))
```

```
    except StopIteration:
```

```
        break #空语句，什么都不做
```



1  
2  
3  
4

# 迭代器

`for i in x`

- 在for循环中，Python将自动调`x.__iter__()`获得迭代器`p`，自动调用`next(p)`获取元素，并且检查`StopIteration`异常，碰到就结束for循环。碰不到就无限循环

# 迭代器

```
class MyRange:

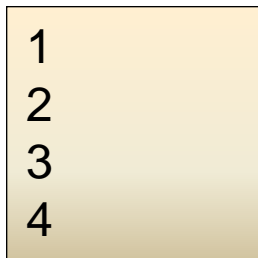
    def __init__(self, n):

        self.idx = 0

        self.n = n

    def __iter__(self):

        return self
```



1  
2  
3  
4



# 迭代器

```
def __next__(self):  
    if self.idx < self.n:  
        val = self.idx  
        self.idx += 1  
        return val  
    else:  
        raise StopIteration() #抛出异常
```

```
for i in MyRange(5):  
    print(i)  
  
print ([i for i in MyRange(5)])
```

0  
1  
2  
3  
4  
[0, 1, 2, 3, 4]

# 迭代器

```
x = MyRange(4)
print([i in i for x])
print([i in i for x])
```

不支持多次迭代!!!

```
[0, 1, 2, 3, ]
[]
```

# 迭代器

## ➤ 容器和迭代器分开解决多次迭代问题

```
class MyRange:
```

```
    def __init__(self, n):
```

```
        self.n = n
```

```
    def __iter__(self):
```

```
        return MyRangeIterator(self.n)
```

# 迭代器

- 容器和迭代器分开解决多次迭代问题

```
class MyRangeIterator:

    def __init__(self, n):

        self.i = 0

        self.n = n

    def __iter__(self):

        return self
```

# 迭代器

```
def __next__(self):  
    if self.i < self.n:  
        i = self.i  
        self.i += 1  
        return i  
    else:  
        raise StopIteration()
```

```
x = MyRange(5)  
  
print([i*i for i in x])  
  
print([i for i in x])
```

```
[0, 2, 4, 9, 16]  
[0, 1, 2, 3, 4]
```

# 生成器(generator)

- 生成器是一种延时求值对象，内部包含计算过程，真正需要时才完成计算

```
a = (i*i for i in range(5)) #a是生成器，其内容并未生成
print(a)                  # <generator object <genexpr> at 0x02436C38>
for x in a:
    print(x,end=" ")
# 0 1 4 9 16
```

# 生成器(generator)

- 生成器是一种延时求值对象，内部包含计算过程，真正需要时才完成计算

```
matrix = ((i*3+j for j in range(3)) for i in range(3))  
# matrix 是生成器，其元素也是生成器  
for x in matrix:  
    for y in x:  
        print(y,end=" ") # 0 1 2 3 4 5 6 7 8
```

# 生成器(generator)

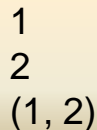
- `yield`关键字用来定义生成器 (Generator)，可以当`return`使用，从函数里返回一个值
- 使用了 `yield` 的函数被称为生成器 (generator)。当函数被调用的时候，并不执行函数，而是返回一个迭代器(iterator)
- 如果`x`是一个生成器被调用时的返回值 (迭代器)，则`next(x)`执行生成器中的语句，直到碰到`yield`语句为止，并且返回 `yield`语句中的对象。如果碰不到`yield`语句函数就结束了，则抛出异常



# 生成器

```
def test_yield():  
    yield 1  
    yield 2  
    yield (1,2)
```

```
a = test_yield()  
while True:  
    try:  
        print(next(a))  
    except StopIteration:  
        break
```



1  
2  
(1, 2)

# 生成器

```
def h():  
    print ('To be brave')  
    yield 5
```

```
x = h() #无输出  
print(next(x))  
next(x) #引发异常
```

To be brave

5

Traceback (most recent call last):

File "D:\studypython\t.py", line 7, in <module>

next(x)

StopIteration

# 生成器

- 使用 yield 实现斐波那契数列:

```
def fibonacci(n): # 生成器函数 - 求斐波那契数列前n项
    a, b, counter = 0, 1, 0
    while counter <= n:
        yield a
        a, b = b, a + b
        counter += 1
```

# 生成器

- 使用 yield 实现斐波那契数列:

`f = fibonacci(10)` # `f` 是一个迭代器, 由生成器返回生成

```
while True:
```

```
    try:
```

```
        print (next(f), end=" ")
```

```
    except StopIteration:
```

```
        break
```

0 1 1 2 3 5 8 13 21 34 55

迭代器+生成器的一个优点就是它不要求事先准备好整个迭代过程中所有的元素。迭代器仅仅在迭代至某个元素时才计算该元素, 而在这之前或之后, 元素可以不存在或者被销毁。这个特点使得它特别适合用于遍历一些巨大的或是无限的集合, 比如几个G的文件, 或是斐波那契数列等等。这个特点被称为延迟计算或惰性求值(Lazy evaluation)。

# 生成器

- 对比类的写法:

```
class Fib:
```

```
    def __init__(self, max): #求max以内Fib数
        self.max = max
```

```
    def __iter__(self):
        self.a = 0
```

```
        self.b = 1
```

```
        return self
```

```
    def __next__(self):
```

```
        fib = self.a
```

```
        if fib > self.max:
```

```
            raise StopIteration
```

```
        self.a, self.b = self.b, self.a + self.b
```

```
        return fib
```

```
for n in Fib(10): #10以内Fib数
    print (n,end="")
```

0 1 1 2 3 5 8

# 生成器

- 用send向yield语句传送数据，没有执行next或send(None)前，不能send(x) (x非None)

```
def h():  
    print ('hello')  
    m = yield 5    #m的值要从外部获得  
    print ("m=", m)  
    d = yield 12  
    print ('d=' ,d )  
    yield 13  
c = h()    #无输出  
x = next(c) #等价于c.send(None)  
print("x=",x)  
y = c.send("ok")  
print("y=",y)  
z = c.send("good")  
print("z=",z)
```

```
hello  
x= 5  
m= ok  
y= 12  
d= good  
z= 13
```

# 生成器

- 用send向yield语句传送数据，没有执行next或send(None)前，不能send(x) (x非None)

```
def h():  
    print ('hello')  
    m = yield 5    #m的值要从外部获得  
    print ("m=", m)  
    d = yield 12  
    print ('d=' ,d )  
    yield 13  
c = h()    #无输出  
x = next(c) #相当于c.send(None)  
print("x=",x)  
y = next(c)  
print("y=",y)  
z = c.send("good")  
print("z=",z)
```

```
hello  
x= 5  
m= None  
y= 12  
d= good  
z= 13
```

# 生成器

## ➤ 生成器表达式和列表解析区别

```
lst = [1,2,3,4]
i = (x+1 for x in lst) #生成器表达式,不需要生成结果列表,延时求值
print(i)
print([x+1 for x in lst]) #列表解析,生成了结果列表
while True:
    try:
        print (next(i))
    except StopIteration:
        pass
#以后不可对 i 再进行迭代了
```

```
<generator object <genexpr> at
0x03A63210>
[2, 3, 4, 5]
2
3
4
5
```



# 生成器

## ➤ 用生成器实现map

```
def myMap(func, iterable):  
    for arg in iterable:  
        yield func(arg)
```

```
names = ["ana", "bob", "dogge"]
```

```
x = myMap(lambda x: x.capitalize(), names)
```

```
print(x)
```

```
print(list(x))
```

```
for name in x: #无输出，因为已经不能对x进行迭代了
```

```
    print(name)
```

```
<generator object myMap at 0x039531C0>  
['Ana', 'Bob', 'Dogge']
```

# 生成器

## ➤ 用生成器求全排列

```
def perm(items):  
    n = len(items)  
    for i in range(len(items)):  
        v = items[i:i+1]  
        if n == 1:  
            yield v  
        else:  
            rest = items[:i] + items[i+1:]  
            for p in perm(rest):  
                yield v + p
```

# 生成器

## ➤ 用生成器求全排列

```
a = perm('abc')
print(next(a))
print(next(a))
print("****")
for b in a:
    print (b)
```

```
abc
acb
****
bac
bca
cab
cba
```

# 闭包(closure)

```
def func(x):  
    def g(y): #g是一个闭包（能维持某个外部变量值的函数）  
        return x+y  
    return g
```

```
f = func(10)  
print(f(4))      #=>14  
print(f(5))      #=>15  
f = func(20)  
print(f(4))      #=>24  
print(f(5))      #=>25
```

# eval函数

➤将字符串看作python表达式并求值，返回其值

```
print(eval("3+2"))  
eval("print(8) ")  
print(eval("'hello'"))
```

=>

5

8

hello

# compile和exec

➤ eval有返回值, exec没有

```
exec ('a=10')
```

```
#execfile(r'c:\test.py') 执行python程序
```

```
print(a)
```

```
str = "for i in range(0,10): print (i,end = ' ')"
```

```
c = compile(str, '', 'exec')          # 编译
```

```
exec(c)          #=>0 1 2 3 4 5 6 7 8 9
```

```
x=4
```

```
y=5
```

```
str2 = "3*x + 4*y"
```

```
c2 = compile(str2, '', 'eval')
```

```
result = eval ( c2 )
```

```
print(result)
```

## eval和exec可以指定作用域

```
a = 10
b = 20
def func(x):
    return x+1
scope={} #用字典当作用域
scope['a'] = 3
scope['b'] = 4
scope['func'] = lambda x:x*x
print(eval("a+b")) #=>30
print(eval("a+b",scope)) #=>7
#print(eval("a+b+c",scope)) error,c无定义
print(eval("func(7)")) #=>8
print(eval("func(7)",scope)) #=>49
exec("print(a)") #=>10
exec("print(a)",scope) #=>3
```

# 反射(reflction)

➤ 反射指的是获取并使用对象的信息

```
class A():
    def __init__(self,x):
        self.x = x
    def func(self):
        print("x=",self.x)

a = A(12)
print(dir(a)) #列出对象所属类的属性, 方法
if hasattr(a,'x'): #判断对象有无属性或方法名为 'x'
    setattr(a,'x','hello,world')

print(getattr(a,'x'))
if hasattr(a,'func'):
    getattr(a,'func')() # 调用 a.func
```



# 反射(reflction)

输出:

```
['__class__', '__delattr__', '__dict__',  
 '__dir__', '__doc__', '__eq__', '__format__',  
 '__ge__', '__getattribute__', '__gt__',  
 '__hash__', '__init__', '__le__', '__lt__',  
 '__module__', '__ne__', '__new__', '__reduce__',  
 '__reduce_ex__', '__repr__', '__setattr__',  
 '__sizeof__', '__str__', '__subclasshook__',  
 '__weakref__', 'func', 'x']  
hello,world  
x= hello,world
```

# 异常处理

➤ `try ... except`

```
try:
```

```
    f = open("ssr.txt", "r")
```

```
except:
```

```
    print("error")
```

```
try:
```

```
    print(aa)
```

```
except Exception as e:
```

```
    print(e)    # name aa is not defined
```

# 异常处理

➤ `try ... except`

```
try:
```

```
    print(aa)
```

```
except IOError as e:
```

```
    print(e)
```

```
except NameError as e:
```

```
    print(e)                # name 'aa' is not defined
```

```
except Exception as e: #捕获一切错误
```

```
    print(e)
```

# 异常处理

➤ `try ... finally`

```
try:
```

```
    print(aa)
```

```
except:
```

```
    pass
```

```
finally:
```

```
    print("done")
```

# 不论有无异常都会执行

# 文件操作

## ➤ 文本文件读写

```
outfile = open("c:\\tmp\\test.txt","w") # w 写 r 读 a 追加
outfile.write("this\nis\na\ngood\ngame")
outfile.close()

infile = open("c:\\tmp\\test.txt","r")
data = infile.read() #读取文件全部内容
print(data)
infile.close()

infile = open("c:\\tmp\\test.txt","r")
for line in infile:
    print(line,end=" ")
infile.close()
```

this  
is  
a  
good  
game  
this  
is  
a  
good  
game

# 文件操作

## ➤ 文件打开模式

**r**

以读方式打开文件，可读取文件信息。

**w**

以写方式打开文件，可向文件写入信息。如文件存在，则清空该文件，再写入新内容

**a**

以追加模式打开文件（即一打开文件，文件指针自动移到文件末尾），如果文件不存在则创建

**r+**

以读写方式打开文件，可对文件进行读和写操作。

**w+**

消除文件内容，然后以读写方式打开文件。

**a+**

以读写方式打开文件，并把文件指针移到文件尾。

# 文件操作

## ➤ 二进制文件读写

```
import sys

from struct import *

tfile = open("c:\\tmp\\test.dat", "wb")

tfile.write(pack('I',1))  #把1当作一个 int写入
tfile.write(pack('I',2))
tfile.write(pack('I',3))
tfile.write(pack('I',4))

tfile.close()
```

# 文件操作

```
infile = open("c:\\tmp\\test.dat", "rb")  
x = infile.read(4)  
x = infile.read(4)  
k = unpack('I', x)[0]    #把x解为一个int  
print(k)                 # 2  
infile.close()
```

pack和unpack格式还有 'f', 'd', 's', 'b' 等



# 文件操作

## ➤ 例:读写学生记录文件

```
class Student:
```

```
    def __init__(self,*info): # 兼容无参构造
```

```
        if info == ():
```

```
            return
```

```
        self.name,self.Id,self.age,self.gpa,self.height = \
```

```
            info[0],info[1],info[2],info[3],info[4]
```

```
    def writeToFile(self,f):
```

```
        bytes=struct.pack('ddi20si',self.gpa,self.height,self.Id, \
```

```
                           self.name.encode("gbk"),self.age)
```

```
        f.write(bytes)
```

```
# 'ddi20si' 表示 : 2个double + 1个int + 20个byte + 1个int
```

```
def readFromFile(self,f):  
    bytes = f.read(44)      #读到文件尾也不会抛出 EOFError异常  
    if bytes == b'':       #读到文件尾  
        return False  
  
    a = struct.unpack('ddi20si',bytes)  # a is a tuple  
    self.gpa,self.height,self.Id,self.name,self.age = \  
        a[0],a[1],a[2],rtrim(str(a[3],encoding="gbk")),a[4]  
    return True  
  
def __str__(self):  
    return "(" + self.name + "," + str(self.Id) + "," \  
        + str(self.age) + "," \  
        + str(self.height) + "," + str(self.gpa) + ")"
```

```
def rtrim(s): # 去除字符串s后面的\x00
    pos = s.find("\x00")
    if pos == -1:
        return s
    else:
        return s[:pos]

ls = [Student("Tom",1078,19,3.7,1.5), \
      Student("Jack",1079,20,3.6,1.8), \
      Student("Marry",1080,21,3.4,1.7)]

f = open("students.dat","wb")

for x in ls:
    x.writeToFile(f)

f.close()
```

```
st = Student()
f = open("students.dat", "rb")
while st.readFromFile(f):
    print(st)
f.close()
f = open("students.dat", "r+b")
f.seek(44+20, os.SEEK_SET)
bytes=struct.pack('20s', "Bob".encode("gbk"))
f.write(bytes)
f.seek(44, os.SEEK_SET)
st.readFromFile(f)
print(st)
```

(Tom,1078,19,1.5,3.7)  
(Jack,1079,20,1.8,3.6)  
(Marry,1080,21,1.7,3.4)  
(Bob,1079,20,1.8,3.6)

# 命令行参数

```
import sys
```

```
for x in sys.argv:
```

```
    print(x)
```

```
'''
```

```
python hello.py this is "hello world"
```

```
argv[0] = 'hello.py'
```

```
argv[1] = 'this'
```

```
argv[2] = 'is'
```

```
argv[3] = 'hello world'
```

```
'''
```

# dir 列出类的方法

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

```
print(dir(l))
```

```
['__add__', '__class__', '__contains__', '__delattr__',  
 '__delitem__', '__dir__', '__doc__', '__eq__', '__format__',  
 '__ge__', '__getattribute__', '__getitem__', '__gt__',  
 '__hash__', '__iadd__', '__imul__', '__init__', '__iter__',  
 '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',  
 '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',  
 '__rmul__', '__setattr__', '__setitem__', '__sizeof__',  
 '__str__', '__subclasshook__', 'append', 'clear', 'copy',  
 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',  
 'sort']
```

# dir 列出类的方法

```
class A:
```

```
    def fun(x):
```

```
        pass
```

```
print(dir(A))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',  
  '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',  
  '__hash__', '__init__', '__le__', '__lt__', '__module__',  
  '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',  
  '__setattr__', '__sizeof__', '__str__', '__subclasshook__',  
  '__weakref__', 'fun']
```

# 文件夹处理

转自: <http://www.jb51.net/article/48001.htm>

- 1.得到当前工作目录，即当前Python脚本工作的目录路径: `os.getcwd()`
- 2.返回指定目录下的所有文件和目录名:`os.listdir()`
- 3.函数用来删除一个文件:`os.remove()`
- 4.删除多个目录: `os.removedirs (r“c: \python”)`
- 5.检验给出的路径是否是一个文件: `os.path.isfile()`
- 6.检验给出的路径是否是一个目录: `os.path.isdir()`
- 7.判断是否是绝对路径: `os.path.isabs()`
- 8.检验给出的路径是否真地存:`os.path.exists()`
- 9.返回一个路径的目录名和文件名:`os.path.split()` eg  
`os.path.split('/home/swaroop/byte/code/poem.txt')` 结果: `('/home/swaroop/byte/code', 'poem.txt')`
- 10.分离扩展名: `os.path.splitext()`
- 11.获取路径名: `os.path.dirname()`
- 12.获取文件名: `os.path.basename()`
- 13.运行shell命令: `os.system()`
- 14.读取和设置环境变量:`os.getenv()` 与 `os.putenv()`



# 文件夹处理

- 15.给出当前平台使用的行终止符:os.linesep Windows使用'\r\n', Linux使用'\n'而Mac使用'\r'
- 16.指示你正在使用的平台: os.name 对于Windows, 它是'nt', 而对于Linux/Unix用户, 它是'posix'
- 17.重命名: os.rename (old, new)
- 18.创建多级目录: os.makedirs (r“c: \python\test”)
- 19.创建单个目录: os.mkdir ( “test”)
- 20.获取文件属性: os.stat (file)
- 21.修改文件权限与时间戳: os.chmod (file)
- 22.终止当前进程: os.exit ( )
- 23.获取文件大小: os.path.getsize (filename)

# python标准库（模块）

转自：<http://www.cnblogs.com/ribavnu/p/4886472.html>

- 文本

1. string: 通用字符串操作

2. re: 正则表达式操作

3. difflib: 差异计算工具

这个模块提供的类和方法用来进行文件或文件夹差异化比较，它能够生成文本或者html格式的差异化比较结果，如果需要比较目录的不同，可以使用[filecmp](#)模块。

4. textwrap: 文本折行

用来将一段英文文本折行表示成比较美观的每行不超过 n 个字符的文本。

5. unicodedata: Unicode字符数据库

6. stringprep: 互联网字符串准备工具

7. readline: GNU按行读取接口

8. rlcompleter: GNU按行读取的实现函数

# python标准库（模块）

- 二进制数据

9. struct: 将字节解析为打包的二进制数据

10. codecs: 注册表与基类的编解码器

# python标准库（模块）

- 数据类型

11. datetime: 基于日期与时间工具

12. calendar: 通用月份函数

13. collections: 容器数据类型

1) namedtuple(): 生成可以使用名字来访问元素内容的tuple子类

```
from collections import namedtuple  
mytuple = namedtuple('mytuple', ['x', 'y', 'z'])  
a = mytuple(3, 5, 7)  
print(a.x) # 3
```

2) deque: 双端队列，可以快速的从另外一侧追加和推出对象

3) Counter: 计数器，主要用来计数

```
a = Counter('ababc')  
b = Counter('12234aaaabd')  
print(a.most_common(3)) # [('a', 2), ('b', 2), ('c', 1)]  
print(b.most_common(3)) # [('a', 4), ('2', 2), ('3', 1)]
```

# python标准库（模块）

## 13. collections: 容器数据类型

4) OrderedDict: 有序字典(按插入顺序排序)

5) defaultdict:

带有默认值的字典(查询的键值不存在时返回默认值, 默认值可以为空list、空dict)

14. collections.abc: 容器虚基类

15. heapq: 堆算法

16. bisect: 在排好序的list中进行二分查找和插入

17. array: 高效数值数组

18. weakref: 弱引用

19. types: 内置类型的动态创建与命名

20. copy: 浅拷贝与深拷贝

21. pprint: 格式化输出

22. reprlib: 交替repr()的实现

# python标准库（模块）

- 数学

23. numbers: 数值的虚基类

24. math: 数学函数

25. cmath: 复数的数学函数

26. decimal: 定点数与浮点数计算

27. fractions: 有理数

28. random: 生成伪随机数

# python标准库（模块）

- 函数式编程

29. **itertools**: 为高效循环生成迭代器,生成无穷序列等

```
import itertools
for key, group in itertools.groupby('AAABBBCCAAA'):
    print(key, list(group))
A ['A', 'A', 'A']
B ['B', 'B', 'B']
C ['C', 'C']
A ['A', 'A', 'A']
```

30. **functools**: 可调用对象上的高阶函数与操作

31. **operator**: 针对函数的标准操作

# python标准库（模块）

- 文件与目录

32. **os.path**: 通用路径名控制

33. **fileinput**: 从多输入流中遍历行

34. **stat**: 解释stat()的结果

35. **filecmp**: 文件与目录的比较函数

36. **tempfile**: 生成临时文件与目录

37. **glob**: Unix风格路径名格式的扩展

38. **fnmatch**: Unix风格路径名格式的比对

39. **linecache**: 文本行的随机存储

40. **shutil**: 高级文件操作,复制文件,文件夹等

41. **macpath**: Mac OS 9路径控制函数



# python标准库（模块）

- 持久化

42. pickle: Python对象序列化

43. copyreg: 注册机对pickle的支持函数

44. shelve: Python对象持久化

45. marshal: 内部Python对象序列化

46. dbm: Unix“数据库”接口

47. sqlite3: 针对SQLite数据库的API 2.0

# python标准库（模块）

**pickle**: Python对象序列化

```
import pickle
```

```
class A:
```

```
    def __init__(self,x,y):
```

```
        self.x = x
```

```
        self.y = y
```

```
obj = {'a':[1,2,3], 'b':12, (1,2):30.5}
```

```
f = open("obj.dat", "wb")
```

```
pickle.dump(obj,f,2)
```

```
pickle.dump(A(10,20),f,2)
```

```
f.close()
```

## python标准库（模块）

```
f = open("obj.dat","rb")
obj2 = pickle.load(f)
print(obj2)    # {(1, 2): 30.5, 'a': [1, 2, 3], 'b': 12}
obj2 = pickle.load(f)
print(obj2.x,obj2.y)    # 10 20
f.close()
```

# python标准库（模块）

- 压缩

- 48. zlib: 兼容gzip的压缩
- 49. gzip: 对gzip文件的支持
- 50. bz2: 对bzip2压缩的支持
- 51. lzma: 使用LZMA算法的压缩
- 52. zipfile: 操作ZIP存档
- 53. tarfile: 读写tar存档文件

# python标准库（模块）

- **zlib** 压缩字符串

```
import zlib
s = "hello word, 0000000000000000aba00000000000000000"
print(len(s)) # 45
c = zlib.compress(bytes(s, 'utf-8'))
print(len(c)) # 28
d = zlib.decompress(c)
print(str(d,encoding="utf-8"))
# hello word, 0000000000000000aba00000000000000000
```

# python标准库（模块）

- `gzip` 压缩和解压文件

```
content = "hello,北京大学"
f = gzip.open('file.txt.gz', 'wb')
f.write(bytes(content,"utf-8"))
f.close()

f = gzip.open('file.txt.gz', 'rb')
content = f.read()
f.close()
print(str(content,encoding="utf-8"))
# hello,北京大学
```

# python标准库（模块）

- **gzip** 压缩现有文件

```
f_in = open('test.txt', 'rb')
f_out = gzip.open('test.txt.gz', 'wb')
f_out.writelines(f_in)
f_out.close()
f_in.close()
```

# python标准库（模块）

- 文件格式化

54. csv: 读写CSV文件

55. configparser: 配置文件解析器

56. netrc: netrc文件处理器

57. xdrlib: XDR数据编码与解码

58. plistlib: 生成和解析Mac OS X .plist文件



# python标准库（模块）

- 加密

59. hashlib: 安全散列与消息摘要

60. hmac: 针对消息认证的键散列

# python标准库（模块）

## • 操作系统工具

### 61. os: 多方面的操作系统接口

62. io: 流核心工具

### 63. time: 时间的查询与转化

64. argparse: 命令行选项、参数和子命令的解析器

65. optparser: 命令行选项解析器

66. getopt: C风格的命令行选项解析器

67. logging: Python日志工具

68. logging.config: 日志配置

69. logging.handlers: 日志处理器

70. getpass: 简易密码输入

71. curses: 字符显示的终端处理

72. curses.textpad: curses程序的文本输入域

73. curses.ascii: ASCII字符集工具

74. curses.panel: curses的控件栈扩展

75. platform: 访问底层平台认证数据

76. errno: 标准错误记号

77. ctypes: Python外部函数库

# python标准库（模块）

- 并发

78. **threading**: 基于线程的并行

79. multiprocessing: 基于进程的并行

80. concurrent: 并发包

81. concurrent.futures: 启动并行任务

82. subprocess: 子进程管理

83. sched: 事件调度

84. queue: 同步队列

85. select: 等待I/O完成

86. dummy\_threading: threading模块的替代（当\_thread不可用时）

87. \_thread: 底层的线程API（threading基于其上）

88. \_dummy\_thread: \_thread模块的替代（当\_thread不可用时）

# python标准库（模块）

- 进程间通信

89. **socket**: 底层网络接口

90. **ssl**: socket对象的TLS/SSL填充器

91. **asyncore**: 异步套接字处理器

92. **asynchat**: 异步套接字命令/响应处理器

93. **signal**: 异步事务信号处理器

94. **mmap**: 内存映射文件支持

# python标准库（模块）

- 互联网

95. email: 邮件与MIME处理包

96. json: JSON编码与解码

97. mailcap: mailcap文件处理

98. mailbox: 多种格式控制邮箱

99. mimetypes: 文件名与MIME类型映射

100. base64: RFC 3548: Base16、Base32、Base64编码

101. binhex: binhex4文件编码与解码

102. binascii: 二进制码与ASCII码间的转化

103. quopri: MIME quoted-printable数据的编码与解码

104. uu: uuencode文件的编码与解码

# python标准库（模块）

- HTML与XML

105. html: HTML支持

106. html.parser: 简单HTML与XHTML解析器

107. html.entities: HTML通用实体的定义

108. xml: XML处理模块

109. xml.etree.ElementTree: 树形XML元素API

110. xml.dom: XML DOM API

111. xml.dom.minidom: XML DOM最小生成树

112. xml.dom.pulldom: 构建部分DOM树的支持

113. xml.sax: SAX2解析的支持

114. xml.sax.handler: SAX处理器基类

115. xml.sax.saxutils: SAX工具

116. xml.sax.xmlreader: SAX解析器接口

117. xml.parsers.expat: 运用Expat快速解析XML

# python标准库（模块）

- 互联网协议与支持

118. webbrowser: 简易Web浏览器控制器

119. cgi: CGI支持

120. cgitb: CGI脚本反向追踪管理器

121. wsgiref: WSGI工具与引用实现

122. urllib: URL处理模块

123. urllib.request: 打开URL连接的扩展库

124. urllib.response: urllib模块的响应类

125. urllib.parse: 将URL解析成组件

126. urllib.error: urllib.request引发的异常类

127. urllib.robotparser: robots.txt的解析器

128. http: HTTP模块

129. http.client: HTTP协议客户端

# python标准库（模块）

- 互联网协议与支持

130. ftplib: FTP协议客户端

131. poplib: POP协议客户端

132. imaplib: IMAP4协议客户端

133. nntplib: NNTP协议客户端

134. smtplib: SMTP协议客户端

135. smtpd: SMTP服务器

136. telnetlib: Telnet客户端

137. uuid: RFC4122的UUID对象

138. socketserver: 网络服务器框架

139. http.server: HTTP服务器

140. http.cookies: HTTPCookie状态管理器



# python标准库（模块）

- 互联网协议与支持

141. http.cookiejar: HTTP客户端的Cookie处理

142. xmlrpc: XML-RPC服务器和客户端模块

143. xmlrpc.client: XML-RPC客户端访问

144. xmlrpc.server: XML-RPC服务器基础

145. ipaddress: IPv4/IPv6控制库

# python标准库（模块）

- 多媒体

146. audioop: 处理原始音频数据

147. aifc: 读写AIFF和AIFC文件

148. sunau: 读写Sun AU文件

149. wave: 读写WAV文件

150. chunk: 读取IFF大文件

151. colorsys: 颜色系统间转化

152. imghdr: 指定图像类型

153. sndhdr: 指定声音文件类型

154. ossaudiodev: 访问兼容OSS的音频设备

# python标准库（模块）

- 国际化

155. gettext: 多语言的国际化服务

156. locale: 国际化服务

- 编程框架

157. turtle: Turtle图形库

158. cmd: 基于行的命令解释器支持

159. shlex: 简单词典分析

- Tk图形用户接口

160. tkinter: Tcl/Tk接口

161. tkinter.ttk: Tk主题控件

162. tkinter.tix: Tk扩展控件

163. tkinter.scrolledtext: 滚轴文本控件

- 其他库.....

# python编写Windows桌面应用

用 wxPython 开发界面

用 py2exe 打包成exe