Python2.5 学习笔记

阿左 [1]　　Nobody [2]

January 12, 2013

[1]感谢读者
[2]感谢国家

Contents

List of Figures

List of Tables

Abstract

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

摘要

Python 之禅

优美胜于丑陋，明了胜于晦涩；简洁胜于复杂，复杂胜于凌乱；扁平胜于嵌套，错落有致胜于密密匝匝。可读性不容轻视。

即便觉得应该为实用性而放弃简单纯粹的时候，“存在特例”也不能成为违反上述规则的借口。

不要让任何错误悄悄溜过，除非你确定不想让人看到报错。当存在多种可能时不要冒险猜测，尽量找到那唯一且明确的答案。

虽然这在一开始并不容易，毕竟你不是 Python 之父。

做也许好过不做，但不假思索就动手还不如不做。难以理解的实现方式绝对糟糕；简章易懂的实现方式往往不错。命名空间是一种绝妙的理念，让我们多加利用！

Part I

语法基础

Chapter 1

模块

1.1 模块与路径

在 linux 环境下查找 python 的安装目录：

```
whereis python
```

在 python 交互环境中查看 python 路径:

```
sys.path
```

1.2 导入模块

1.2.1 导入模块

导入模块使用import 语句：

```
import module01
```

当模块被导入后，会包含源文件的目录信息：< 目录 >/< 文件名 >.< 扩展名 >。

1.2.2 重新导入模块

一个模块只能有一个实例，导入一个模块以后不能再次导入。当一个模块的代码被修改以后必须使用reload 语句重新加载模块才能生效：

```
reload module01
```

reload 只会重新导入写出的那个模块，不会自动导入相关的模块。

1.2.3 部分导入模块

如果只导入模块中的部分变量，则使用from 语句：

```
from module01 import a, b, c
```

from 语句不导入模块，只复制模块中的变量到本地。复制模块中全部变量的例子：

```
from module01 import *
```

import 与from 语句都都有赋值效果，例如：

```
from module1 import a, b, c
```

等价于：

```
import module1

a = module1.a
b = module1.b
c = module1.c
```

1.2.4 防止变量被 from 复制

方法一：下划线开头的变量不会被复制，如："_X"。

方法二：在顶层变量"__all__"列表中的成员不会被复制出。如：

```
__all__ = ['Error', 'encode']
```

1.2.5 模块别名

导入了模块以后，使用时要加上包名，所以有必要使用简写：

```
import longname as name
```

等价于：

```
import longname
name = longname
del longname
```

from 语句也可以用简写：

```
from mod import longnmae as name
```

1.2.6 判断是否是直接执行

模块有__name__ 属性。如果是被导入的，值为模块的名字；如果是被直接调用的，值为：'__main__'。可以通过它判断程序是否是被导入的：

```
if __name__ =  '__main__' :
  # in main
```

1.3 直接执行脚本文件

可以不加载模块而是直接以脚本文件的方式执行：

```
execfile('mymodule.py')
```

1.4 包的使用

1.4.1 建立自己的包

包放在 python 检索的目录下。包目录与包下的每一级都要有__init__.py 文件（内容可以为空）表明这是一个包，__init__.py 文件中的语句都会在导入时被执行。例：

```
# dir1/__init__.py
print 'dir1'
x = 1
```

```
# dir1/dir2/__init__.py
print 'dir2'
y = 2
```

```
# dir1/dir2/mod.py
print 'mod.py'
z = 3
```

1.4.2 使用自己的包

导入自己包的例子：

```
% python
>>> import dir1.dir2.mod
dir1
dir2
mod.py

>>> import dir1.dir2.mod

>>> reload(dir1)
dir1

>>> reload(dir1.dir2)
dir2

>>> dir1.x
1
>>> dir1.dir2.y
2
>>> dir1.dir2.mod.z
3
```

Chapter 2

核心数据类型

在 python 的核数据类型中，数字、字符串、元组这三个对象是不可变的。

2.1 基本数字类型

数字对象是不可变的，许多对它的操作会生成一个新的对象返回，而不是改变原来的对象。

```
123, 5, 0                          # 同语言的长整数C
999999999999L                      # 无限制长度
1.23, 3.14e-10, 4E210, 4.0e+210 # 浮点数类似语言双精度数C
0177, 0x13e, 0X3e                  # 八进制与十六进制
3+4j, 3.0+4.0j, 3J                 # 复数，实数加上复数
```

格式化数字：

```
num = 1 / 3.0

print num
```

```
print '%e'    % num            # string formating
print '%2.2f' % num            # string formating
print '%o %x %X' % (64,64,255) # '100 40 FF'
```

常用操作：加（+）、减（-）、乘（*）、除（/）、下取整（//）、乘方（**）、位移（>>）等：

```
print 1 /  3         # 0
print 1 // 3         # 0
print 1.0 / 3        # 0.33333333333
print 1.0 // 3       # 0.0
```

调用内置的函数进行强制类型转换：

```
print int(3.1415)    # 3
print float(3)       # 3.0
print long(4)        # 4

print oct(64)        # 0100
print hex(64)        # 0x40

print int('0100')    # 100
print int('0100', 8) # 64
print int('0x40',16) # 64
```

2.1.1 常用数学模块

```
# coding=utf-8
import random
import math

print random.random()                # 0.323492834792
print random.randint(1,10)           # 3
```

```
print random.choice([1,2,3,4,5])      # 2
print random.choice(['aa','bb','cc'])  # cc

print math.e
print math.pi
print math.sqrt(85)
print math.sin(2*math.pi / 180)

print abs(-42)       # 42
print pow(2,4)       # 16
print round(2.567)   # 3.0
print round(2.567,2) # 2.57
```

2.2 十进制小数类型

集合类型 set 可以支持一般的数学集合操作：

```
# coding=utf-8

import decimal

d = decimal.Decimal('3.141')

print d+1

print decimal.Decimal('0.1') + decimal.Decimal('0.1') \
  + decimal.Decimal('0.1')
print decimal.Decimal('1') / decimal.Decimal('7')

# set precision
decimal.getcontext().prec = 4
print decimal.Decimal('1') / decimal.Decimal('7')
```

2.3 序列类型的共同点

字符串、列表、元组都属于序列类型类型，会支持一些共同的操作。

索引操作：

```
s = 'Spam'
print len(s)          # length
print s[0]            # index start from 0
print s[1]            #
print s[-1]           # first from end
print s[len(s)-1]     # first from end
print s[-2]           # second from end
```

分片操作：

```
print s[1:3]
print s[1:]
print s[:3]
print s[:-1]
print s[:]
print s[::2]          # set step
print s[::-1]         # backward
```

连接与重复操作会生成新的对象：

```
print s + '^_^'       # create new object, s still the same
print s * 3           # create new object, s still the same
```

2.4 字符串类型

字符串也是一种序列类型，所以支持所有的序列操作。

字符串对象是不可变的，许多对它的操作会生成一个新的对象返回，而不是改变原来的对象。

2.4.1 常用方法

python 中的字符串使用双引号和单引号都是一样的。而且空值不会像在 C 语言里中断字符串。

三重引号可以按原文格式生成文本。可以用来注释大段的代码。

```
# coding=utf-8

# use oct and hex
print str(42)
print int("42")
print str(3.14159265)
print float("1.234E-10")

print ord('\n')       # show 10, because'\n' binary value is 0x10
print ord('s')          # 115
print chr(115)          # s

s = '\001\002\x03'
print s
print len(s)            # 3

# user unicode
print u'spam'
print u'ab\x20cd'       # 1-byte char : ab cd
print u'ab\u0020cd'     # 2-byte char : ab cd
print u'ab\U00000020cd' # 4-byte char : ab cd

# do not escape
print r'c:\new\text.txt'

# corss lines
print """ threre is
cross lines...
```

```
29 hahahah....  """
30
31 print ''' threre is
32 cross lines...
33 hahahah....  '''
34
35
36 line = '   aaa,bbb,ccc,ddd,eee   '
37 print line.lstrip()
38 print line.rstrip()
39 print line.upper()
40 print line.isalpha()       # False
41 print 'abc'.endswith('c')  # True
42
43 # join up the string
44 print 'aaa' + "bbb"    # aaabbb
45 print 'aaa' "bbb"      # aaabbb
46
47 # create a list
48 ll =  'aaa',"bbb"      # ('aaa','bbb')
49
50 # string format
51 exclamation = "Ni"
52
53 print "%d %s %d you" % (1, 'spam', 4)
54 # use turple to set more element into one place
55 print "%s -- %s -- %s" % (42, 3.14159, [1,2,3])
56 # format by dict
57 print "%(n)d %(x)s" % {"n":1, "x":"spam"}
58
59 s = 'hello a hello b hello c'
60 print s.find('hello')           # offsit is 0
61 print s.replace('hello', 'bye') # bye a bye b bye c
62
```

```
63 # break string to list
64 l = list('abcdefg')
65 print l                # ['a', 'b', 'c', 'd', 'e', 'f', 'g']
66 # join list to string
67 print ''.join(l)       # abcdefg
68 print '-'.join(l)       # a-b-c-d-e-f-g
69
70 # split string
71 print 'aa bb cc'.split()    # ['aa','bb','cc']
72 print 'aa,bb,cc'.split(',') # ['aa','bb','cc']
```

2.5 列表

列表也是一种序列类型，所以支持所有的序列操作。而且列表是可变类型。

虽然列表没有固定大小，但是还是会越界。越界会抛出 IndexError。

列表常用的特定操作：

```
1  l = [123, 'spam', 1.23, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2
3  l.index(1)
4
5  l.append('NI') # [123, 'spam', 1.23, 'NI']
6  l.extend([5,6,7])
7  l.insert(1,88)
8
9  l.pop(2)       # 1.23 # [123, 'spam', 'NI']
10 l.remove(2)
11 del l[1]
12 del l[1:2]
13
14 l[0] = 5
```

```
l[3:5] = [66,77,88]

range(4)       # create new list [1,2,3,4]
xrange(0,4)    # xrange(4)

l.sort()
l.reverse()
```

列表可以进行嵌套，并且经常被用在多维数组上：

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

print m[1]    # [4,5,6] print row 2
print m[1][2] # 6       print row 2, item 3

# show col 3 use list comprehension expression
print [row[1] for row in m]                    # [2,5,8]
print [row[1]+1 for row in m]                  # [3,6,9]
print [row[1] for row in m if row[1] % 2 == 0 ] # [2,8]

print [m[i][i] for i in [0,1,2]]          # [1,5,9]
print [c*2 for c in 'span']               # ['ss','pp','aa','mm']
```

2.6 元组

元组也是一种序列类型，所以支持所有的序列操作。

元组对象是不可变的，许多对它的操作会生成一个新的对象返回，而不是改变原来的对象。

新建一个元组的操作：

```
tuple(l)
```

2.7 字典

创建一个字典的方法：

```
d = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
d2 = dict.fromkeys(['a','b']) # {'a':None, 'b':None}
d3 = dict(zip(d.keys(),d.values()))
d4 = dict(name='skinner', age=18)

d.copy()

# create new dict and add attr
d = {}
d['name'] = 'Bob'
d['job'] = 'Dev'
d['age'] = 25
```

字典的常用方法：

```
d = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
d['a']
d['a'] = 33
d['a'] += 1

d2.update(d3)  # update d2's value as d3's same key value

d.pop('a')
del d['c']

len(d)
```

```
12
13 d.keys()
14 d.values()
15 d.items()
```

字典中的键不一定要用字符串，可以用任何不可变对象。不能读取一个字典中不存在的键，但可以给不存在的键设一个默认值：

```
1 'f' in d
2 d.has_key('f')
3 d.get('f', 'default')
```

字典中的项目可以是不同的类型：

```
1 # nesting
2 rec = {'name': {'first':'Jade', 'last':'Shan'},
3     'job': ['dev','mgr'],
4     'age': 25}
5
6 print rec['name']['last']     # Shan
7 print rec['job'][-1]          # mgr
```

对字典内容进行排序：

```
1 ks = d.keys()                 # ['a','c','b']
2 ks.sort()                     # sort keys
3 for key in ks:
4   print key, ' => ', d[key]
5
6 for key in sorted(d):
7   print key, ' => ', d[key]
```

用整数来作为字典的键，模拟一个不会越界的列表：

```
1 d = {}
2 d[99] = 'spam'
```

用字典实现稀疏数据结构：

```
mtx = {}
mtx[(2,3,4)] = 88
mtx[(7,8,9)] = 99

print mtx.get((2,3,4),0)
print mtx.get((7,8,9),0)
print mtx.get((0,8,9),0)
```

字典可以直接迭代：

```
for key in dic :
  print key, dic[key]
```

2.8 集合类型

集合类型 set 可以支持一般的数学集合操作：

```
x = set('spam')
y = set(['h','a','m'])

print x,y   # set(['a', 'p', 's', 'm']) set(['a', 'h', 'm'])
print x & y # set(['a', 'm'])
print x | y # set(['a', 'p', 's', 'h', 'm'])
print x - y # set(['p', 's'])

engineers = set(['bob','sue','ann','vic'])
managers  = set(['tom','sue'])

print engineers & managers # set(['sue'])
print engineers - managers # set(['vic', 'bob', 'ann'])
print engineers | managers # set(['vic', 'sue', 'tom',
```

```
15                                  # 'bob', 'ann'])
```

2.9 bool 类型与空

bool 类型实际是在以定制后的逻辑来显示整数的 1 和 0。

None 对象表示空

```
1 print 1>2           # False
2 print bool('spam')  # True
3 print None          # None
```

常见对象的布尔值状态：

```
1 "spam"    True
2 ""        False
3 []        False
4 {}        False
5 1         True
6 0.0       False
7 None      False
```

常见操作：

```
1 X and Y
2 X or  Y
3 not x
```

2.10 type 类型

再次强调 python 中的类型信息与变量无关，类型是关联在对象上的。

```
l = [None] * 100

print type(l)                   # <type 'list'>
print type(type(l))             # <type 'type'>

print type(l) == type([])       # True
print type(l) == list           # True
print isinstance(l, list)       # True
```

2.11 对象的副本

当一个复杂的对象内部有一个引用是它自己的话，就形成了循环引用的情况。python 会把循环引用的对象打印为[...]。但是程序在处理的时候还是会造成无限循环。

2.11.1 对象的副本

要为序列类对象为了建立一个用于修改的副本，用全部分片是最方便的方法：

```
l1 = [1,2,3,4,5]
l2 = l1[:]
l2[0] = 0
print l1          # [1, 2, 3, 4, 5]
print l2          # [0, 2, 3, 4, 5]
```

对于字典对象，有成员方法copy：

```
dic1 = {'a':1,'b':2,'c':3}
dic2 = dic1.copy()
dic2['a'] = 0
print dic1        # {'a': 1, 'c': 3, 'b': 2}
print dic2        # {'a': 0, 'c': 3, 'b': 2}
```

标准库中的 copy 可以拷贝任意对象，分只复制顶层与深层复制：

```
import copy

l1 = [1,2,3,4,5]
dic1 = {'a':1,'b':2,'c':l1}
dic2 = copy.copy(dic1)
dic2['a'] = 99
dic2['c'][0] = 99
print dic1          # {'a': 1, 'c': [99, 2, 3, 4, 5], 'b': 2}
print dic2          # {'a': 99, 'c': [99, 2, 3, 4, 5], 'b': 2}

l1 = [1,2,3,4,5]
dic1 = {'a':1,'b':2,'c':l1}
dic2 = copy.deepcopy(dic1)
dic2['a'] = 99
dic2['c'][0] = 99
print dic1          # {'a': 1, 'c': [1, 2, 3, 4, 5], 'b': 2}
print dic2          # {'a': 99, 'c': [99, 2, 3, 4, 5], 'b': 2}
```

序列的重复会生成新的序列：

```
l = [4,5,6]
x = l * 4    # new list
y = [l] * 4  # reference l 4 time

print x # [4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
print y # [[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]

l[1] = 0
print x # [4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
print y # [[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

2.11.2 对象的相等与同一

判断两个对象是否相等用“==”操作符，判断两个对象是不是同一个对象用“is”操作符：

```
l1 = [1,2,3]
l2 = [1,2,3]
print l1 == l2     # True
print l1 is l2     # False

x = 42
y = 42
print x == y       # True
print x is y       # True
```

在sys 模块中的getrefcount 函数会返回一个对象被引用的次数：

```
import sys
print sys.getrefcount(1)  # 86
print sys.getrefcount(x)  # 11
```

Chapter 3

语句顺序

表达式语句：

```
span(a,b)         # func call
spam.ham(a)       # call attr method
a < b < c         # range test
a==true and b<1   # condition test
```

3.1 if 语句

基本结构：

```
if ... :
  ...
elif ... :
  ...
else
  ...
```

3.1.1 推荐风格

对于过长的语句可以使用“\”换行：

```
if a==b and b==c \
   and c==d :
   print 'all equal!'
```

但推荐还是用“()”包起来在风格上更好：

```
if (a==b and b==c
   and c==d):
   print 'all equal!'
```

3.2 三元表达式

```
<normal> if <condition> else <other>
```

相同的表现形式，如逻辑判断的短路会只执行一个：

```
((x and y) or z)
```

可以用 bool 作为下标[0] 或[1]：

```
[z,y][bool(x)]
```

3.3 用字典实现判断

python 中的字典也可以部分实现 if-else 的功能。因为字典不但可以用函数作为成员，而且取值操作时，可以指定找不到值的默认返回值：

```
myDict.get('aaa','no value');  # default value if key not exist
```

要注意的是，字典的判断和逻辑操作不同，没有短路特性。

3.4 while 循环语句

```
while ... :
  ...       # 执行的语句
  ...       # 遇到 break 跳出整个循环
  ...       # 遇到 continue 回到开头进行
  ...       # pass 语句什么也不做，仅在语法上点一个位置
else :
  ...       # 只有在循环完全执行后才会运行（没有遇到）break
```

3.5 for 循环语句

```
for ... in ... :
  ...       # 执行的语句
  ...       # 遇到 break 跳出整个循环
  ...       # 遇到 continue 回到开头进行
  ...       # pass 语句什么也不做，仅在语法上点一个位置
else :
  ...       # 只有在循环完全执行后才会运行（没有遇到）break
```

range 函数会展开数字为列表：

```
range(5)
range(len(list))
range(0,5,2)      # [0,2,4]
```

3.5.1 for 循环例子

用xreadlines() 一行一行地读入文本文件，防止文件过大：

```
for line in open('log.txt').xreadlines():
  print line
```

用推荐自动用速度快而内存利用最合理的方法：

```
for line in open('log.txt'):
  print line
```

用range() 函数生成列表的方法：

```
range(5)        # [0,1,2,3,4]
range(2, 5)     # [2,3,4]
range(0, 10, 2) # [0,2,4,6,8]
```

结合range() 函数与 for 循环：

```
for x in range(5, -5, 2):
  print x

for i in range(len('abc')):
  print 'abc'[i]
```

3.6 循环条件不能赋值

python 中的循环条件不同于 C 语言，不能有赋值操作：

```
while( null != (x=next()) )
{ ... }
```

python 中可以用另一种形式写：

```
while True:
  x = next()
  if not x : break
```

3.7 迭代器

3.7.1 通用迭代器

python 中所有的对象都可以被迭代，迭代的工具是通用迭代器next()。当没有可迭代的项时接收StopIteration 异常决定离开。

3.7.2 文件迭代

文件迭代可以使用包装过了reanline()。它会的文件结尾返回空字符串'':

```
file = open('log.txt')
file.readline()
file.readline()
file.readline()
...
file.readline()        # return '' at end of file
```

```
for line in open('aa.txt'):
  ....

for line in open('aa.txt').readlines():
  ....

for line in open('aa.txt').xreadlines():
  ....
```

3.7.3 常见的迭代器用法

处理每个成员

```
list = [line.upper() for line in lines]
```

map() 函数，声明用指定的方法处理每个成员

```
map(str.upper, lines)
```

成员处理

```
'3' in lines      # 是否包含成员
sort(lines)       # 排序
sum([1,2,3,4,5])  # 15
any(['a','','c']) # True
all(['a','','c']) # False
```

转换类函数

```
list(line)        # 转为列表
tuple(line)       # 转为元组
```

合并多个可迭代序列

```
zip('ABC','abc','123')
# result (('A','a','1'), ('B','b','2'), ('C','c','3'))
```

enumerate() 把列表解析为“下标”与“值”的形式

```
items = enumerate('abc')
items.next()              # result is (0,'a')
```

结合 enumerate() 与 for 循环

```
for (index, value) in enumerate('abc')
```

3.8 列表解析

列表解析以中括号围起的单行 for 循环的形式编写：

```
[... for ... in ...]
```

一般来说，列表解析会比用 for 循环更快。原因是在底层有优化，它是以 C 运行的。在速度上“列表解析”快于“map”快于“for 循环”。

3.8.1 列表解析与 for 循环结合的例子

读取文件并且去除最后的换行符：

```
lines = [line.rstrip() for line in open('log.txt')]
```

组合 for 与 if 读取以“p”开头的行：

```
lines = [line.rstrip() for line in open('log.txt') if line[0]=='p']
```

列表解析也可以嵌套：

```
[x+y for x in 'abc' for y in '123']
# result is ['a1','a2','a3' ... 'c1','c2','c3']
```

列表解析与多维数组结合：

```
del l[1]
del l[1:2]

l[0] = 5
l[3:5] = [66,77,88]

range(4)      # create list [1,2,3,4]
xrange(0,4)   # xrange(4)

l.sort()
l.reverse()

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

Chapter 4

函数

4.1 函数定义

函数都有返回值。没有 return 或 yield 语句的函数会返回一个 None。

python 中，函数也是一个对象。def 语句是实时执行的，当执行到def 语句时就会定义一个函数对象，然后赋值给指定函数名变量：

```
1 def myAdd(a, b):
2   return a+b
```

以上代码生成了一个函数对象，并且把这个对象赋值给了变量myAdd。可以通过变量myAdd 调用这个函数：

```
1 myAdd(1,2)
```

也可以把函数再赋值给另一个变量：

```
1 otherFunc = myadd
2 otherFunc(1,2)
```

函数对象的生成与赋值是在运行时发生的。所以也可以在运行时创建函数：

```
if a>3:
  otherFunc = funcA
if a<3:
  otherFunc = funcB
```

4.2 参数传递

函数参数没有参数类型限制，只要参数支持对应的操作就可以。如果参数不支持对应的操作会抛出异常。

4.2.1 默认参数

函数参数列表支持name=value 形式的默认值：

```
def myAdd(a=1, b=2):
  return a+b
```

函数的默认参数被认为是静态的，不会每次调用都生成一个新的对象。例：

```
def myfunc(x=[]):
  x.addend(1)
  print x

myfunc([2])       # [2,1]
myfunc()       # [1]
myfunc()       # [1,1]
myfunc()       # [1,1,1]
myfunc()       # [1,1,1,1]
```

为了让每次建立新的默认值，可以在函数内写赋值语句：

```
def myfunc(x=None):
  if x is None:
    x = []
  x.addend(1)
  print x
```

4.2.2 可变参数

python 支持可变参数。当定义函数时，* 表示列表、** 表示字典：

```
def func(a, * pargs, ** kargs):
    print a, pargs, kargs

func(1,2,3,x=1,y=2)           # 1 (2, 3) {'y': 2, 'x': 1}
```

当调用函数时，* 表示把列表打散成多个参数、** 表示把字典打散作为参数：

```
lst = (1,2,3,4,5)
dic = {'a':1,'b':2,'c':3}

func(*lst)                    # 1 (2, 3, 4, 5) {}
func(**dic)                   # 1 () {'c': 3, 'b': 2}

def func(a,b,c):
  print a

func( *(1,2,3) )          # OK
func(  (1,2,3) )          # err
func(  2 )                # err
```

混合使用参数的格式：

```
def func(a,b,c,... j=k,l=m,... *p,*q,... **x,**y...)
```

```
        ^         ^           ^         ^
        normal    key-value   seq       dict
```

4.3 变量的作用域

4.3.1 全局变量

用global 关键字声明要用的是全局变量：

```
x = 5
def func():
  global x
  x = 99
```

4.3.2 内置变量

内置作用域变量要导入后才能使用：

```
import __builtin__
```

4.4 lambda 表达式

lambda 表达式中不能出现语句，只能用表达式。例：

```
f = lambda a,b : a+b
f(1,2)
```

用表达式替代常用语句。例：

```
lbd = lambda x : sys.stdout.write(x+'\n')
lbd('aaa')

lbd = lambda x : map(sys.stdout.write, x)
lbd(['aaa\n','bbb\n','ccc\n'])

lbd = lambda x : [sys.stdout.write(line) for line in x]
lbd(['aaa\n','bbb\n','ccc\n'])
```

4.5 闭包

一个函数的成员变量可以被定义在这个函数内部的内部函数访问。

在早期不支持闭包的版本中，内部函数不能访问外部函数的变量；只能通过参数的默认值来取得外部函数变量的值：

```
def func1():
  x = 88
  def func2(x=x):    # 作为默认参数赋值
    print x
  func2()
```

在使用闭包的情况下：

```
def func1():
  x = 88
  def func2():
    print x
  return func2

action = func1() # call func1()
action()         # call func2()
```

要注意的一点是：函数只有在第一次被调用时存在。

所以下面的例子中，变量i 的值永远是 4。而不是随循环从 0 增长到 4：

```
def func1():
  acts = []
  for i in range(5):
    acts.append(lambda x: i ** x)
  return acts
```

在这种情况下，只能使用默认参数：

```
def func1():
  acts = []
  for i in range(5):
    acts.append(lambda x,i=i: i ** x)
  return acts
```

4.6 结合函数处理序列

4.6.1 过滤（filter）

过滤大于 0 的元素：

```
filter( (lamabda x : x>0), [-2,-1,0,1,2,3,4])
# result: [1,2,3,4,]
```

4.6.2 汇总（reduce）

所有的元素加起来：

```
reduce( (lamabda x,y : x+y), [1,2,3,4])   # 10
```

所有的元素乘起来：

```
reduce( (lamabda x,y : x*y), [1,2,3,4])   # 24
```

在 python3 中可能移除 reduce，但自己实现一个并不难：

```
def myreduce(func, seq):
  tally = seq[0]
  for i in seq[1:]:
    tally = function(tally,i)
  return tally
```

Chapter 5

OOP 与类

一个模块只能有一个实例，当一个模块的代码被修改以后必须重新加载模块才能生效；类可以同时创建多个实例（即对象）。

5.1 最简单的 Python 类

Python 的类模型相当动态，类与实例只是命名空间对象。

可以仅用 pass 作为占位语句生成一个没有任何成员的 class，这样的 class 仅仅是一个空的命名空间对象。可以在以后通过赋值给这个类加上新的成员：

```
1  class C1: pass              # Empty class
2                              # as an namespace object
3
4  a = C1()
5  a.name = "morgan"           # a.name is "morgan"
6
7  C1.name = "Class C1"        # C1.name is "C1"
8  b = C1()                    # b.name is "C1"
```

```
                          # a.name still is "morgan"

def upperName(self):
  print self.name.upper()

C1.showName = upperName     # add func to class
a.showName()
b.showName()
```

5.2 类与实例的基本概念

类与类的实例都是对象。类是一个对象，该类的每个实例又各对应了一个关联到该类对象的实例对象。

像调用函数一样调用类对象会创建该类的实例对象。每个实例对象根据具体类的创建属性并获得自己的命名空间，每个成员都有自己的实例。

对象的数据成员（无论是数据还是函数）在没有被第一次赋值之前，都不能被访问（就像是没有被声明的变量一样）。同样地也可以简章地通过赋值给变量增加一个成员。

class 语句会创建一个类对象并赋值给变量名。class 语句块内的语句会创建数据对象和方法对象赋值给类的成员变量。类的成员变成只属于该类，不属于该类的实例，但可以被所有的实例共享访问。

```
class Ca:
  v = 42

a = Ca()   # a.v = 42
b = Ca()   # b.v = 42

Ca.v = 55  # Ca.v = 55
           # a.v = 55
       # b.v = 55
```

```
10
11 a.v = 11   # Ca.v = 55
12            # a.v = 11
13        # b.v = 55
```

类中的方法被所有函数共享。通过实例调用方法时，会把这个实例对象作为第一个参数 self 传递给方法。

例子：

```
1 class c1():
2   count = 0
3   def show(self):
4     print 'c1.show()'
```

类的构造函数是__init__；析构函数是__del__：

```
1 class Life:
2   def __init__(self, name='unknow'):
3     print 'hello ', name
4     self.name = name
5   def __del__(self):
6     print 'goodbye ', self.name
7
8 brian = Life('Brian')     # prints: hello Brian
9 brian = 'loretta'         # prints: goodbye Brian
```

实例对象的__class__ 属性记录了这个实例的类对象。

```
1 print a.__class__           # class of instance
```

5.3 类的成员

5.3.1 属性与方法

python 中对于类和对象，无论数据还是方法都作为变量处理。对象的数据成员（无论是数据还是函数）在没有被第一次赋值之前，都不能被访问（就像是没有被声明的变量一样）。同样地也可以简单地通过赋值给变量增加一个成员。

```
class Employee():
  def __init__(self, name):   # __init__ defing construter
    self.name = name
  def showName(self):
    print self.name

morgan = Employee("Morgan")
morgan.showName()

morgan.showName = "new name"    # set value
morgan.nickName = "Jade"        # create new variable
```

类对象与实例对象的__dict__ 属性是大多数基于类的对象的命名空间字典。

```
print Employee.__dict__.keys()  # 大多数基于类的对象的命名空间字典
print morgan.__dict__.keys()    # 大多数基于类的对象的命名空间字典
```

5.3.2 类属性与对象属性

修改了类的属性以后会影响到所有的类实例：

```
class C:
  a = 1

print C.a     # is 1
```

```
o = C()
print o.a     # is 1

C.a = 2
print o.a    # is 2, also change too

j = X()
print j.a   # is 2
```

5.3.3 方法与对象的绑定

对于以下的类：

```
class Spam:
  def doit(self, message):
    print message
```

可以通过新建对象调用：

```
obj = Spam()
obj.doit('hello world')
```

已经绑定的方法可以再赋值给一个变量，效果和对象点号调用一样：

```
x = obj.doit
x('hello world')
```

但是类中的方法是没有绑定到对象的，要传入对象：

```
t = Spam.doit
t(obj, 'hello world')
```

同理，如果要引致类中的函数，相同的规则也适用于类的方法：

```
class Eggs:
  def m1(self, n):
    print n

  def m2(self):
    x = self.m1  # bound to object
    x(42)

Eggs().m2()
```

5.4 类的继承

类对象的成员__bases__ 是超类构成的元组

```
print c3.__bases__                # 超类的元组
```

通过Super 调用超类：

```
class Super:
  def __init__(self):
    pass

class Sub(Super):
  def __init__(self):
    Super.__init__(self):
      pass
```

5.4.1 多重继承

子类会继承父类的成员。如果当多重继承时遇到重名成员时顺序就很重要，优先级按声明继承时从左到右顺序：

```
class c1():
  myname = 'aaa'
  def show(self):
    print 'c1.show()'
  def helloC1(self):
    print 'c1.helloC1()'

class c2():
  def show(self):
    print 'c2.show()'
  def helloC2(self):
    print 'c2.helloC2()'

class c3(c1, c2):                # 超类写在括号中，支持多重继承
  def show(self):
    print 'c3.show()'

c1.myname                        # c1.myname is 'aaa'
c3.myname                        # c3.myname is 'aaa'
```

对象成员赋值后不影响类成员的值：

```
i1 = c3()                        # i1.myname is 'aaa'
i2 = c3()                        # i2.myname is 'aaa'

i1.myname = 'this is i1'         # i1.myname is 'this is i1'
i2.myname = 'this is i2'         # i2.myname is 'this is i2'
print c1.myname                  # c1.myname still is 'aaa'
print c3.myname                  # c3.myname still is 'aaa'

i1.show()                        # 调用重写的方法
i1.helloC1()                     # 调用到父类的方法
i1.helloC2()                     # 调用到父类的方法
```

虽然顺序很重要，但是也可以手动指定要调用哪一个：

```
class A:
  def __repr__(self): pass
  def other(self): pass

class B:
  def __repr__(self): pass
  def other(self): pass

class C(A, B):
  __repr__ = A.__repr__
  other = B.other
```

5.4.2 压缩成员变量名

常见的变量名重复的情况：

```
class C1:
  def meth1(self): self.x = 88
  def meth2(self): print self.x

class C2:
  def metha(self): self.x = 99
  def methb(self): print self.x

class C3(C1, C2): pass

I = C3()   # only 1 x in I !!!
```

在这样的情况下，x 只有一个。两个类的四个方法读写的都是同一个 x 成员。这个 x 是值取决最后是哪一个方法调用的它。

在__class__ 语句中以两个下划线开头的变量__verb 会扩展成包含类名的新变量名_classname__verb。这样就不会和同一层次中其他类所创建的类似变量名相冲突。

```
class C1:
  def meth1(self): self.__x = 88
  def meth2(self): print self.__x

class C2:
  def metha(self): self.__x = 99
  def methb(self): print self.__x

class C3(C1, C2): pass

I = C3()   # two x in I: _C1__x and _C2__x

I.meth1()
I.meth2()  # result is 88

I.metha()
I.methb()  # result is 99
```

5.4.3 继承的应用场景

一般由超类提供默认的方法或是等待实现的方法：

```
class Super:
  def method(self):
    print 'Super.method()'           # default behavior
  def delegate(self):
    self.action()                   # expected to defined
```

子类可以不去实现超类：

```
class Inheritor(Super):
```

```
2   pass                                    # inherit method verbatim
```

子类可以把超类的方法完全覆盖掉：

```
1 class Replacer(Super):
2   def method(self):
3     print 'Replacer.method()'         # replace method completely
```

子类可以扩展超类的方法：

```
1 class Extender(Super):
2   def method(self):                     # extend method behavior
3     print 'Extender.method() start'
4     Super.method(self)
5     print 'Extender.method() end'
```

子类可以实现超类有待实现的方法：

```
1 class Provider(Super):                    # fill in required method
2   def action(self):
3     print 'Provider.action'
```

在迭代中创建类：

```
1 if __name__ == '__main__':
2   for clazz in (Inheritor, Replacer, Extender):
3     print '\n' + clazz.__name__ + '...'
4     clazz().method()
5   print '\nProvider...'
6   p = Provider()
7   p.delegate()
```

5.5 命名空间

5.5.1 命名空间范围

python 中作用域是由源代码中赋值语句位置来决定的，不会受到导入关系的影响。通过下面的例子总结了 python 的命名空间：

```
# coding=utf-8

X = 11              # Golbal(module) name/attribute

def f():
  print X         # Access global X(11)

def g():
  X = 22          # Local (function) varible
  print X

class C:
  X = 33          # Class attribute (C.X)
  def m(self):
    X = 44      # Local variable in method
    self.X = 55 # Instance attribute (instance.x)

if __name__ == '__main__':
  print X       # 11: module(a.k.a. manynames.X in outside file)
  f()           # 11: global
  g()           # 22: local
  print X       # 11: module name unchanged

  obj = C()     # Make instance
  print obj.X   # 33: class anme inherited by instance

  obj.m()       # Attach attribute name X to instance now
```

```
28   print obj.X   # 55: instance
29   print C.X     # 33: class( a.k.a. obj.X if not X in instance)
30
31 # print C.m.X[26]  # FAILS: only visible in method
32 # print f.X        # FAILS: only visible n function
```

需要注意的是在调用 I.m() 之前实例中是不存在成员变量 X 的。实例的成员变量也是变量，在执行到赋值语句之前是不会存在的。

在外部文件中使用：

```
1  # coding=utf-8
2
3  import manynames
4
5  X = 66
6
7  print X             # 66: the global here
8
9  print manynames.X   # 11: globals become attributes after imports
10
11 manynames.f()       # 11: func f() print manyname.X
12 manynames.g()       # 22: local in other file's function
13
14 print manynames.C.X # 33: attribute of class in other module
15
16 I = manynames.C()
17 print I.X           # 33: still from class here
18
19 I.m()
20 print I.X           # 55: now from instance
```

5.5.2 命名空间字典

无论是模块、对象还是实例，它们的命名空间实际都是以字典形式实现的。这个字典就是对象中的成员__dict__ 变量。

```
>>> class super:
...     def hello(self):
...             self.data1 = 'spam'
...
>>> class sub(super):
...     def hola(self):
...             self.data2 = 'eggs'
...
>>> X = sub()
>>> X.__dict__
{}
>>> X.__class__
<class __main__.sub at 0x7f35b8248ef0>
>>> sub.__bases__
(<class __main__.super at 0x7f35b8248e88>,)
>>> super.__bases__
()
```

以self 为目标赋值会把成员加入到对象，而不是类中：

```
>>> Y = sub()

>>> X.hello()
>>> X.__dict__
{'data1': 'spam'}

>>> X.hola()
>>> X.__dict__
{'data1': 'spam', 'data2': 'eggs'}

```

```
>>> sub.__dict__
{'__module__': '__main__', '__doc__': None, 'hola': <function hola at 0
    x7f35b8201668>}

>>> super.__dict__
{'__module__': '__main__', 'hello': <function hello at 0x7f35b82015f0>, '
    __doc__': None}

>>> Y.__dict__
{}
```

可以用命名空间字典部分实现点号操作，但是字典不带继承搜索：

```
>>> X.__dict__['data1']
'spam'

>>> X.__dict__['data3'] = 'ham'
>>> X.data3
'ham'
```

5.5.3 命名空间链接

每个类的实例，都有成员变量__class__ 连接到它的类；

而在类中有成员变量__bases__ 是一个元组，包含了到超类的链接。

5.6 运算符重载

两头是双下划线的方法名（__方 法 名__）表示对运算符的重载。

5.6.1 重载字符串化方法

当要把实例字符串化时会先使用__str__。这样产生的结果在终端下用户体验较好；如果没有定义，会再调用__repr__，它产生的结果可以作为代码直接重建该对象。

```
class adder:
  def __init__(self, value=0):
    self.data = value
  def __str__(self):
    return '[value is: %s]' % self.data
  def __repr__(self):
    return 'addrepr(%s)' % self.data

x = adder(2)
print x          # [value is: 2]
print str(x)     # [value is: 2]
print repr(x)    # addrepr(2)
```

5.6.2 常用数学方法

__add__ 表示重载+ 运算。

__sub__ 表示重载-运算。

__mul__ 表示重载* 运算。

对于没有定义或是继承的操作符，表示该操作不被支持。

```
class Number():
  def __init__(self, value):
    self.data = value
  def __add__(self, other):
    return C1(self.data + other)
  def __sub__(self, other):
    return C1(self.data - other)
```

```
  def __mul__(self, other):
    return C1(self.data * other)

a = Number(5)       # Number.__init__(a,5)
b = a - 2           # Number.__sub__(a,2)
```

以上的方法中，操作符右边的不能是实例对象。还有一个相反的左边不是对象右边是对象的__radd__ 方法：

```
class Computer:
  def __init__(self, val):
    self.val = val
  def __add__(self,other):
    self.val += other
  def __radd__(self, other):
    self.val = self.val + other

x = Computer(2)
x + 3
print x.val

y = Computer(3)
3 + y
print y.val
```

5.6.3 拦截索引

索引操作可以通过__getitem__ 方法拦截：

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

```

```
x = indexer()
x[2]                    # __getitem__(x,2)  result 4
```

索引操作可以模拟迭代效果：

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

x = stepper()
x.data = "spam"

for item in x:
  print item
```

5.6.4 拦截迭代

迭代操作可以通过__iter__ 方法拦截，python 在遇到迭代环境时会先尝试迭代，如果对象不支持迭代，就会尝试索引运算：

```
class Squares:
  def __init__(self, start, stop):    # save state when created
    self.value = start -1
    self.stop = stop
  def __iter__(self):          # get iterator object on iter()
    return self
  def next(self):              # return a square on each iteration
    if self.value == self.stop:
      raise StopIteration
    self.value += 1
    return self.value ** 2

for i in Squares(1,5):   # for calls iter(), which calls __iter__()
```

```
  print i              # Each iteration calls next()
                       # get result: 1 4 9 16 25
```

__iter__ 只循环一次，循环以后就变为空，每次新的循环就必须重新创建一个新的迭代器对象。

```
x = Squares(1,5)
[n for n in x]            # [1,4,9,16,25]
[n for n in x]            # [] now it's empty
[n for n in Squares(1,5)] # [1,4,9,16,25]
list(Squares(1,3))        # [1,4,9]
```

另外一个例子，迭代时跳过下一下元素：

```
class SkipIterator:
  def __init__(self, wrapped):
    self.wrapped = wrapped  # iter state flg
    self.offset = 0
  def next(self):
    if self.offset >= len(self.wrapped):
      raise StopIteration
    else:
      item = self.wrapped[self.offset]
      self.offset += 2
      return item

class SkipObject:
  def __init__(self,wrapped):
    self.wrapped = wrapped
  def __iter__(self):
    return SkipIterator(self.wrapped)

if __name__ == '__main__':
  skipper = SkipObject('0123456789')
  i = iter(skipper)   # make an iterator on it
```

```
22   print i.next(), i.next(), i.next()
23
24   # in earch nest create new iterator
25   for c in skipper:
26     for d in skipper:
27       print c+d
```

在最后的嵌套的两个for 循环中，每个循环都会建立自己的迭代器，相互之间不会混淆。这相的操作相当于：

```
1  str = 'abcdefg'
2  for x in s[::2]
3    for y in s[::2]
4      print x+y
```

这里迭代与分片的区别是。分片把结果的整个列表放到了内存里，而迭代器每次产一个值；分片会产一个新的对象，并不同于迭代是对原来的对象进行 r 操作。

5.6.5 拦截成员属性

__getattr__ 对成员属性的点号操作时行拦截（多数情况下是对没有定义的成员属性）：

```
1  class empty:
2    height = 189
3    def __getattr__(self,name):
4      if attrname == "age":
5        return 40
6
7  x.age          # 40
8  x.height       # 189
9  x.name         # None
```

其他属性抛出异常：

```
class empty:
  height = 189
  def __getattr__(self,name):
    if attrname == "age":
      return 40
    else:
      raise AttributeError, attrname

x.age          # 40
x.height       # 189
x.name         # exception
```

__setattr__ 对成员属性的赋值操作进行拦截。但有个问题：即使在__setattr__内的self.attr=value 也会再次被拦截，造成无限循环。所以在拦截函数内要通过：self.__dict__['name']=value 来赋值：

```
class AccessControl:
  def __setattr__(self,attr,value):
    if attr == 'age':
      self.__dict__[attr] = value
    else:
      raise AttributeError, attr + ' not allowed'

x = AccessControl()
x.age = 40
```

5.6.6 模拟私有成员

```
class PrivateExc(Exception):pass

class Privacy:
  def __setattr__(self, attrname, value):
```

```
    if attrname in self.privates:
      raise PrivateExc(attrname, self)
    else:
      self.__dict__[attrname] = value

class Test1(Privacy):
  privates = ['age']

class Test2(Privacy):
  privates = ['name','pay']
  def __init__(self):
    self.__dict__['name'] = 'Tom'
```

5.6.7 拦截调用

如果定义了__call__ 方法，实例的调用就会被该方法拦截：

```
class Prod:
  def __init__(self,value):
    self.value = value
  def __call__(self,other):
    return self.value * other

x = Prod(2)   # x.value is 2
x(3)          # return value is 2*3=6
x(4)          # return value is 2*4=8
```

主要的应用场合是作为回调函数调用，比如在 GUI 程序中：

```
class CallBack:
  def __init__(self,color):
    self.color = color
  def show(self):
```

```
5       print 'color: ' + self.color
6
7   a = CallBack('red')
8   b = CallBack('blue')
9
10  b1 = Button(command=a)
11  b2 = Button(command=b)
```

除了绑定对象还可以直接绑定到方法：

```
1   b1 = Button(command=a.show)
2   b2 = Button(command=b.show)
```

lambda 表达式也经常用来实现回调函数的功能：

```
1   c = (lambda color='red' : 'color is: '+color)
2   print c()
```

5.7 类的设计

5.7.1 通用对象工厂

python 中的对象工厂要比强类型语言中简单得多：

```
1   def factory(clazz, *args):
2     return apply(clazz, args)
3
4   class Spam:
5     def doit(self,message):
6       print message
7
8   class Person:
9     def __init__(self, name, job):
```

```
10      self.name = name
11      self.job = job
12
13  obj1 = factory(Spam)
14  obj2 = factory(Person, "Guido", "guru")
```

5.8 静态方法和类方法

最典型的静态方法应用场景是统计一个类产生了多少实例：

```
1  class Spam:
2    count = 0
3    def __init__(self):
4      Spam.count = Spam.count +1
5    def printCount():
6      print "count: " , Spam.count
```

但是这样是行不通的。在调用printCount() 时一定要传入一个实例作为参数，无论你在def 定义函数是有没有写明self。虽然可以给它绑定一个对象，但这样有一个缺点就是一定要通过对象来调用这个函数。

相对方便一点的方法是它移动到类外，作为一个模块下的简单函数。例：

```
1  class Spam:
2    count = 0
3    def __init__(self):
4      Spam.count = Spam.count +1
5
6  def printCount():
7    print "count: ", Spam.count
```

这样类和函数都是模块命名空间下的变量。也不会和其他文件中的变量名冲突：

```
import spam

a = spam.Spam()
b = spam.Spam()
c = spam.Spam()

spam.printCount()  # module.function : result is 3
spam.Spam.count    # module.class.verb : result is 3
```

5.8.1 使用静态方法和类方法

可以通过内置函数staticmethod 和classmethod 来生成静态方法和类方法。这两都都不需要在启用时传入实例作为参数。

```
class Multi:
  def imeth(self,x):    # normal instance method
    print self, x
  def smeth(x):         # static: no instance passed
    print x
  def cmeth(clazz,x):   # Class: get class, not instance
    print clazz, x
  smeth = staticmethod(smeth) # make smeth a static method
  cmeth = classmethod(cmeth)  # make cmeth a class method
```

实例方法、静态方法、类方法的使用：

```
obj = Multi()
obj.imeth(1)         # normal call
Multi.imeth(obj, 1)  # normal call

Multi.smeth(3)       # static call, through class
obj.smeth(4)         # static call, through instance

```

```
Multi.cmethod(5)     # class call, throw class
obj.cmethod(5)       # class call, throw instance
```

5.8.2 函数装饰器

函数包装器可以在函数外面包上层逻辑。如静态函数可以这样写：

```
class C:
  @staticmethod
  def method(): pass
```

相当于前面的静态函数实现：

```
class C:
  def method(): pass
  method = staticmethod(method)
```

实际上装饰器语法可以包上多层逻辑：

```
@A @B @C
def f(): pass
```

相当于前面的静态函数实现：

```
def f(): pass
f = A(B(C(f)))
```

5.8.3 函数装饰器例子

```
class tracer:
    def __init__(self,func):
            self.calls = 0
            self.func = func
```

```
5     def __call__(self, *args):
6             self.calls += 1
7             print 'call %s to %s' % (self.calls, self.func.__name__)
8             self.func(*args)
9
10 @tracer
11 def spam(a, b, c):
12     print a, b, c
```

用__call__ 重载方法替实例实现函数调用接口。因为 spam 函数是通过 tracer 装饰器执行的，所以实际上调用的是类中的__call__ 方法。

```
1 spam(1,2,3)       # call 1 to spam
2                   # 1 2 3
3
4 spam('a','b','c') # call 2 to spam
5                   # a b c
6
7 spam(4,5,6)       # call 3 to spam
8                   # 4 5 6
```

Chapter 6

异常

6.1 捕获异常

```
try:
  statements
except type1:                 # match type
  statements
except type2, data:           # type and data
  statements
except (type3, type4):        # match those type
  statements
except (type3, type4), value: # match type get data
  statements
except:                       # all other type
  statements
else:
  statements
finally:
  statements
```

6.1.1 基于字符串的异常

基于字符串的异常马上就是被淘汰了。

字符串异常的捕获是按照同一对象来匹配的。不是同一个字符串对象即便值相同也不会被捕获到。

6.1.2 基于类的异常

通过__repr__ 或是__str__ 定义异常文本:

```
class MyBad:
    def __repr__(self):
      return "Sorry-my mistake..."
```

6.2 产生异常

```
raise string          # maches same string object
raise string, data    # extra data(default = None)
raise class, instance #
raise instance        # same as: raise ins.__class__, ins
raise                 # re-raise current exception
```

6.3 附加信息

6.3.1 异常类中附加信息

在 except 分句中:

```
except type, instance:
  statement
```

type 和 instance 分别代表异常的类和实例：

```
class FE:
  def __init__(self, line, file):
    self.line = line
    self.file = file

try:
  raise FormatError(42, file='spam.txt')
except FormatError, err
  print 'error at: ', err.file, err.line
```

6.3.2 字符串异常中附加信息

字符串中要额外附加一个对象来添加附加信息：

```
errstr = 'FormatError'

try:
  raise errstr, {'line':42, 'file':'spam.txt'}
except errstr, err:
  print 'error at: ', err['file'], err['line']
```

6.4 获得最新的异常

可以通过sys.exc_info 取得最新引发的异常。返回值是包含type、value、traceback'三个成员的元组；在没有异常的情况下会返回成员是三个None 的无组。

6.5 断言

```
import asserter

def f(x):
  assert x < 0, 'x must be nagative'
  return x ** 2

asserter.f(1)
```

6.6 环境管理器

6.6.1 使用环境管理

使用with-as 环境管理会包装一个异常处理工作:

格式:

```
with expression [as variable]:
  with-block
```

表达式产生的对象一定要是支持环境管理协议的对象，如文件对象已经支持环境管理协议:

```
with open('aa.txt') as myfile:
    for line in myfile:
            print line
```

6.6.2 实现环境管理协议

必须要有__enter__ 和__exit__ 方法。

__enter__ 方法被调用后的返回值会赋值给as 指定的变量。如果没有as 则丢弃这个变量。

如果 with 代码块引发了异常，__exit__(type, value, traceback) 会被调用。如果此方法返回值为 False，会再次抛出异常；否则表示异常已经被处理完毕。

如果 iwth 代码块没有发生异常，__exit__ 方法还是会被调用，其中的 type、value、trackback 参数都会为 None。

例子：

```
1  # coding=utf-8
2
3  class TB:
4    def message(self, arg):
5      print 'running', arg
6    def __enter__(self):
7      print 'starting with block'
8      return self
9    def __exit__(self, exc_type, exc_value, exc_tb):
10     if exc_type is None:
11       print 'exited normally\n'
12     else:
13       print 'raise an exception!', exc_type
14       return False
15
16 with TB() as action:
17   action.message('test 1')
18   print 'reached'
19
20 with TB() as action:
21   action.message('test 2')
22   raise TypeError
23   print 'not reached'
```

Chapter 7

python 文档

7.1 PyDoc 查看文档内容

使用dir(obj) 可以查看对象的所有成员。

使用help() 函数查看帮助:

```
1 help(sys.getrefcount)
```

浏览 HTML 版的文档:

```
pydoc.py -g
或
<pythonDir>/Tools/pydocgui.pyw
```

7.2 程序内文档

文件、类、函数头上的文档字符串会被自动封装为文档。

```
'''
    document text of module
'''

def func(x)
  '''
  document of function
  '''
  pass

class MyClass:
  '''
  document of class
  '''
  pass
```

可以通过它们的__doc__ 成员显示文档的内容：

```
print filename.__doc__
print filename.classname.__doc__
print filename.funcname.__doc__
```

Chapter 8

常用功能

8.1 输入输出

8.1.1 python 中的输出流

```
import sys

sys.stdout.write('hello\n')      # 输出到标准输出
sys.stderr.write('Error...\n')   # 输出到标准错误
```

8.1.2 print 打印输出

print 语句会把对象打印到默认的输出流（标准输出）中：

```
print a, b, b...
```

格式化打印为a => b 的效果：

```
print '%s => %s' % ('a', 'b')
```

8.1.3 重定向 print 到其他输出流

方法一：用指定的输出流替换掉标准输出。这样有一个缺点是每次都要手动地打开与关闭输出流：

```
import sys

sys.stdout = open('aa.txt', 'a');
print 'hello'
sys.stdout.close()
```

方法二：可以在print 语句中指定输出流：

```
import sys

log = open('log.txt', 'w')
print >> log, 'start', 1, 2, 3   # write to log file
log.close()
print >> sys.stderr, "err..."    # write to std err
```

8.2 文件操作

8.2.1 常用的文件操作

```
input  = open('aa.txt')       # 输入文件 r读取(默认)
str = input.read()            # 读取整个文件
str = input.read(n)           # 读取多个字节
str = input.readline()        # 读取多个字节
```

```
lst = input.readlines()

output = open('aa.txt','w')  # 输出文件 w写入
output.write(str)            #
output.writelines(lst)       #
output.flush()
output.close()

anyFile.seek(n)              # move to n
```

8.2.2 文本读写

```
1  # coding=utf-8
2
3  f = open('data.txt','w')
4  f.write('hello\n')
5  f.write('world\n')
6  f.close()
7
8  f = open('data.txt')   # default flag is 'r'
9  bytes = f.read()
10 print bytes
11 print bytes.split()
```

8.2.3 对象的存取

python 的内置函数eval() 可以直接把字符串作为程序代码执行。所以可以用它来把文本变成 python 对象：

```
1  # coding=utf-8
2
```

```
l = [1,2,3]
d = {'a':1, 'b':2, 'c':3}

out = open('obj.txt','w')
out.write(str(l) + "\n")
out.write(str(d) + "\n")
out.flush()
out.close()

infile = open('obj.txt', 'r')
lines = infile.readlines()
infile.close()

objs = [eval(line) for line in lines]
print objs       # [[1, 2, 3], {'a': 1, 'c': 3, 'b': 2}]
```

但是eval() 函数有一个安全隐患：它会执行“任何”python 代码……你懂的……

另一种选择是使用可以用来存取几乎任何 python 对象的 pickle 模块：

```
# coding=utf-8

import pickle

d = {'a':1, 'b':2, 'c':3}

outfile = open('data.txt','w')
pickle.dump(d,outfile)
outfile.close()

infile = open('data.txt')
obj = pickle.load(infile)

print obj              # {'a': 1, 'c': 3, 'b': 2}
```

8.2.4 二进制文件读写

struct 模块能打包并解析二进制文件：

```
# coding=utf-8

import struct

bytes = struct.pack('>i4sh',1,'spam',2)
print bytes

outfile = open('data.bin','wb')
outfile.write(bytes)
outfile.close()

infile = open('data.bin','rb')
data = infile.read()
infile.close()

print data

obj = struct.unpack('>i4sh',data)
print obj
```

8.3 正则表达式

```
# coding=utf-8

import re

match = re.match('Hello[ \t]*(.*)world','Hello  Python world')
print match.group(1)                          #  match 'Python'
```

```
7
8  match = re.match('/(.*)/(.*)/(.*)', '/usr/home/lumberjack')
9  print match.groups()         # ('usr', 'home', 'lumberjack')
10 print len(match.groups())    # 3
11 print match.group(1)         # 'usr'
12 print match.groups()[0]      # 'usr'
```

Part II

常用功能

Chapter 9

文本

9.1 字符与其值的转换

```
# coding=utf-8

print ord('a')    # 97
print chr(97)     # a

print ord(u'\u2020')     # 8224
print repr(unichr(8224)) # u'\u2020'

print map(ord, 'ciao')
# [99, 105, 97 111]
print ''.join(map(chr, range(97,100))) # abc


#将转换成普通的字符串UnicodePython编码:"(encode)"
unicodestring = u"Hello world"
utf8string = unicodestring.encode("utf-8")
```

```
asciistring = unicodestring.encode("ascii")
isostring = unicodestring.encode("ISO-8859-1")
utf16string = unicodestring.encode("utf-16")


#将普通的字符串转换成PythonUnicode: 解码"(decode)"
plainstring1 = unicode(utf8string, "utf-8")
plainstring2 = unicode(asciistring, "ascii")
plainstring3 = unicode(isostring, "ISO-8859-1")
plainstring4 = unicode(utf16string, "utf-16")
```

9.2 判断一个对象是否是字符串

basestring 是 str 类型与 Unicode 类型的共同基类，但是 Python 标准库中的UserString 模块提供的UserString 类不是从它派生出来的。这种情况下可能尝试对象是否能像字符串一样工作（鸭子判断法）来判断，缺点是异常块的执行性能较差。

```
# coding=utf-8

def isString(obj):
  return isinstance(obj, basestring)

print isString('asdf')
print isString(u'\u2020')
print isString(65535)

def isLikeString(obj):
  try:
    obj + ''
  except:
    return False
  else:
```

```
16      return True
17
18  print isLikeString('asdf')
19  print isLikeString(555)
```

9.3 对齐字符串

本来对齐字符串很简单。

```
1   # coding=utf-8
2
3
4   print '|', 'hello'.ljust(50), '|'
5   print '|', 'hellohello'.rjust(50), '|'
6   print '|', 'hello hello hello'.center(50), '|'
7   print '|', '坚持党的基本路线年不变100'.center(50), '|'
```

但是全角字符的显示宽度不一样，要特别处理一下：

```
1   # coding=utf-8
2
3   import unicodedata
4
5   otherDoubleWidthList = \
6     'A B C D E F G H I J K L M N O P Q R S T U V W X Y Z' + \
7     'a b c d e f g h i j k l m n o p q r s t u v w x y z' + \
8     '0 1 2 3 4 5 6 7 8 9' + '[ ]( ){ } ' " ¬ ` ' + \
9     '+ < = > ~ ^ @# % & *__ — ¯ \ | ¦ / ' + \
10    '¢ $ ¥ £' + ',.:;?!'
11  otherDoubleWidthList = otherDoubleWidthList.decode('utf-8')
12
13
14
```

```
15 def countStrWidthCJK(str, ambiwidth=2):
16   '''计算字符串显示长度，能够处理字符宽度CJK
17   ambiwidth: 宽度不定的字符算几个，取值为 1, 2'''
18   if ambiwidth == 2:
19     doubleWidth = ('W', 'A')
20   elif ambiwidth == 1:
21     doubleWidth = ('W',)
22   else:
23     raise ValueError('ambiwidth 取值为 1 或者 2')
24
25   count = 0
26   unicodeStr = str.decode('utf-8')
27   for i in unicodeStr:
28     if (unicodedata.east_asian_width(i) in doubleWidth \
29         or i in otherDoubleWidthList):
30       count += 2
31       continue
32     count += 1
33
34   return count
35
36
37
38 def filterMissDoubleWidthStr(str):
39   '''检查有哪些宽字符被错判断为半字符'''
40   for c in tmpStr.decode('utf-8'):
41     if countStrWidthCJK(c.encode('utf-8')) == 1 :
42       print c, '=', countStrWidthCJK(c.encode('utf-8'))
43
44
45
46 def justCJK(str, width, type='l', fillChar=' '):
47   '''对齐字符串，能够处理字符宽度CJK'''
48   fillWidth = width - countStrWidthCJK(str)
```

```
49    if fillWidth < 0:
50      fillWidth = 0
51
52    if('l' == type):
53      return str + fillChar * fillWidth
54    elif('r' == type):
55      return fillChar * fillWidth + str
56    else:
57      mt = fillWidth % 2
58      si = fillWidth / 2
59      return fillChar * si + str + fillChar * (si+mt)
60
61
62
63  print '|' + justCJK('一三五七九24680',60,'c')  + '|'
64  print '|' + justCJK('A B C D E F G H I',60,'r')  + '|'
65  print '|' + justCJK('a b c d e f g h i',60,'l')  + '|'
66  print '|' + justCJK('0 1 2 3 4 5 6 7 8 9',60,'c')  + '|'
67  print '|' + justCJK('《》[ ]( ){ } ' "',60,'r')  + '|'
68  print '|' + justCJK('0987654321',60,'r')  + '|'
69  print '|' + justCJK(',.:;?!',60,'r')  + '|'
70  print '|' + justCJK('[ ]( ){ } ' " ¬ `',60,'r')  + '|'
71  print '|' + justCJK('+ < = > ~ ^ @# % & *__— ¯ \ | ¦ /',60,'r')  + '|'
72
73  tmpStr = ''
74  filterMissDoubleWidthStr(tmpStr)
```

9.4 拼接字符串

字符串可以直接用加号拼接，不过性能低下：

```
1  list = {'a','b','c','d'}
```

```
2
3 str = ''
4 for c in list:
5   str = str + c
6 print str
```

下面的代码相同与：

```
1 import operator
2 print reduce(operator.add, list, '')
```

用join 方法通过指定的字符串把各个部分拼接起来：

```
1 print ''.join(list)
```

更加复杂的格式可以用格式把字符串方法：

```
1 print '%s %s %s' % ('a','b','c')
```

9.5 反转字符串

反转每一个字符：

```
1 str = '0123456789'
2 print str[::-1]
```

反转单词可以先按空格打散，然后再接起来：

```
1 str = 'Hello world\n Hi\tPython'
2 tmp = str.split()
3 tmp.reverse()
4 tmp = ' '.join(tmp)
```

挤成一行的写法：

```
tmp = ' '.join(str.split()[::-1])
```

用正则来保留原来是用空格还是用其他空白字符分隔的：

```
import re
tmp = re.split(r'(\s+)', str)
tmp.reverse()
tmp = ' '.join(tmp)
print tmp
```

挤成一行的写法：

```
tmp = ' '.join(re.split(r'(\s+)',str)[::-1])
```

9.6 包含与不包含

判断字符（不光是字符串，还有其他的集合）是否属于一个集合：

```
def containsAny(seq, aSet):
  """序列中的成员是否属于某集合"""
  for item in seq:
    if item in aSet: return True
  return False
```

用itertools 可以提高一点性能：

```
序列中的成员是否属于某集合
def containsAny(seq, aSet):
  """序列中的成员是否属于某集合"""
  for item in itertools.ifilter(aSet.__contains__, seq):
    return True
  return False
```

9.7 控制大小写

```
print 'one tWo thrEe'.upper()      # ONE TWO THREE
print 'one tWo thrEe'.lower()      # one two three
print 'one tWo thrEe'.capitalize() # One two three
print 'one tWo thrEe'.title()      # One Two Three
```

9.8 字符串模板替换

```
tmp = string.Template('Value is $value')
print tmp.substitute({'value':5})
print tmp.substitute({'value':'default'})
print tmp.substitute(value=5)
print tmp.substitute(value='default')
```

9.9 一次完成多个替换

可以先根据 dict 的 key 建立了一个正则表达式；然后在re.sub 调用过程中使用了回调函数代替了用于替换的字符串。这样每当匹配时，re.sub 会调用该回调函数，以它的返回值作为替换的字符串：

```
import re

def multipleReplace(text, aDict):
  print '|'.join(map(re.escape, aDict))
  rx = re.compile('|'.join(map(re.escape, aDict)))
  print rx
  def one_xlat(match):
    return aDict[match.group(0)]
```

```
9   return rx.sub(one_xlat, text)
10
11 adic = {'aaa':'111','bbb':'222','ccc':'333'}
12 print multipleReplace('aaa bbb ccc',adic)
```

9.10 检查字符串的结尾

字符串的endwith 函数判断是否以指定字符串结尾。主要的思路是把字符串的endwith作为参数传递给一个中间函数anyTrue。然后用itertools.imap 来映射检查每一项：

```
1 import itertools
2
3 def anyTrue(predicate, sequence):
4   return True in itertools.imap(predicate, sequence)
5
6 def endsWith(s, *endings):
7   return anyTrue(s.endswith, endings)
```

上例中

一个应用的场景是查找图片文件：

```
1 import os
2
3 for filename in os.listdir('.'):
4   if endsWith(filename, '.jpg', '.png', '.gif'):
5     print filename
```

9.11 应用 Unicode

python 中 Unicode 文本和普通的字符串不是同一种对象，不能混合操作。所以要时刻明白你正在处理的文本对象是普通字符串还是 Unicode 字串。一般来说，会用以下措施来

防止错误发生：

在收到外部的文本数据时，应该创建一个 uncode 对象，通过查看 HTTP 头之类的方法来确定所用的编码方式。不然错误的操作会引发 UnicodeDeocdeError。

同理在向外部发送文本之前要用正确的编码把文本转化为字节串，不然有可能发生 UnicodeEncodeError。

9.12 Unicode 与字符串之间的转换

转换时要先确定编码：

```
1  uniStr = u"Hello world"
2
3  rec = uniStr.encode("ascii")
4  print rec
5  rec = unicode(rec,"ascii")
6  print rec
7
8  rec = uniStr.encode("iso-8859-1")
9  print rec
10 rec = unicode(rec,"iso-8859-1")
11 print rec
12
13 rec = uniStr.encode("utf-8")
14 print rec
15 rec = unicode(rec,"utf-8")
16 print rec
```

9.13 在标准输出中打印 Unicode

通过 cocdes 模块把 sys.stdout 流用转换器包装起来。如果你要输出的终端以 iso-8859-1 编码显示字符：

```
1 import codecs, sys
2
3 sys.stdout = codecs.lookup('iso8859-1')[-1](sys.stdout)
```

上面的代码把sys.stdout 绑定到一个使用 Unicode 输入和 ISO-8859-1 输入的流。这样的好处是不会改变sys.stdout 上原来的编码。

Python 会尝试确认“终端”所使用的编码（并不一定正确）并把编码的名字存放到sys.stdout.encoding 中作为一个属性。

9.14 在 XML 与 HTML 中使用 Unicode

Chapter 10

文件

10.1 基础

一次读取整个文件会占用大量内存：

```
for line in input.readlines():
  process(line)
```

可以一次读取一行：

```
for line in input:
  pricess(line)
```

对于二进制文件在read 调用时指定读取 N 个字节。不指定的话默认会读取剩下的全部内容。

包装读取文件与处理：

```
def scanner(file, handler):
  for line in file:
    handler(line)
```

把具体的实例传递进去：

```
from myutils import scanner

def firstword(line):
  print line.split()[0]

file = open('data.txt')

scanner(file, firstword)
```

现在这个scanner 的范围只能用来扫描文件。为了扩展应用范围到也能扫描字符串，我们可以用 StringIO 或是性能更好的 cStringIO 把字符串封装为文件：

```
from cStringIO import StringIO
from myutils import scanner

def firstword(line):
  print line.split()[0]

string = StringIO('one\nTwo xxx\nthree\n')
scanner(string, firstword)
```

其实还可以有个更加通用的方案就是用包含迭代器的类把它包装起来：

```
from cStringIO import StringIO
from myutils import scanner

def firstword(line):
  print line.split()[0]

string = StringIO('one\nTwo xxx\nthree\n')
scanner(string, firstword)
```

Part III

其他扩展