Python2.5 学习笔记

阿左 [1]　　Nobody [2]

May 22, 2012

[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 Pytho 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

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 >。

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
```

如果只导入模块中的部分变量，则使用from 语句：

```
1 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：

```
1 from module01 import *
```

import 与from 语句都都有赋值效果，例如：

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

等价于：

```
1 import module1
2
3 a = module1.a
4 b = module1.b
5 c = module1.c
```

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：

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

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。

支持的操作：加（+）、减（-）、乘（*）、除（/）、乘方（**）。

2.1.1 常用数学模块

```
1 import math
2 import random
3
4 print math.pi
5 print math.sqrt(85)
6
7 print random.random()
8 print random.choice([1,2,3,4,5])
```

2.2 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：

```
# coding=utf-8

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
```

2.3 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。

```
print s.find('pa')
print s.replace('pa','^_^')
print s.upper()
print s.isalpha()

line = '   aaa,bbb,ccc,ddd,eee   '
print line.lstrip()
print line.rstrip()
print line.split(',')

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

2.4 列表

列表也是一种序列类型，所以支持所有的序列操作。

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：

```
# coding=utf-8

l = [123, 'spam', 1.23]

print l.append('NI') # None
print l              # [123, 'spam', 1.23, 'NI']
print l.pop(2)       # 1.23
print l              # [123, 'spam', 'NI']

print l.sort()       # Nome
print l              # [123, 'NI', 'spam']
```

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]]
```

```
4
5  print m[1]    # [4,5,6] print row 2
6  print m[1][2] # 6       print row 2, item 3
7
8  # show col 3 use list comprehension expression
9  print [row[1] for row in m]                    # [2,5,8]
10 print [row[1]+1 for row in m]                  # [3,6,9]
11 print [row[1] for row in m if row[1] % 2 == 0 ] # [2,8]
12
13 print [m[i][i] for i in [0,1,2]]          # [1,5,9]
14 print [c*2 for c in 'span']               # ['ss','pp','aa','mm']
```

2.5 字典

不能读取一个字典中不存在的键：

```
1  # coding=utf-8
2
3  d = {'food':'spam', 'quantity':4, 'color':'pink'}
4  print d.has_key('f')          # False
5
6  print d['food']
7  d['quantity'] += 1
8  print d['quantity']
9
10 # create new dict and add attr
11 d = {}
12
13 d['name'] = 'Bob'
14 d['job'] = 'Dev'
15 d['age'] = 25
16
17 print d
18
19 # nesting
20 rec = {'name': {'first':'Jade', 'last':'Shan'},
```

```
    'job': ['dev','mgr'],
    'age': 25}

print rec['name']['last']     # Shan
print rec['job'][-1]          # mgr
```

对字典内容进行排序：

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

ks = d.keys()                 # ['a','c','b']
ks.sort()                     # sort keys
for key in ks:
  print key, ' => ', d[key]

for key in sorted(d):
  print key, ' => ', d[key]
```

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。

```
# coding=utf-8

f = open('data.txt','w')
f.write('hello\n')
f.write('world\n')
f.close()

f = open('data.txt')   # default flag is 'r'
bytes = f.read()
print bytes
print bytes.split()
```

2.7 集合类型

集合类型 set 可以支持一般的数学集合操作：

```
# coding=utf-8

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'])
```

2.8 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

```
# coding=utf-8

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

2.9 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')
```

```
6
7 print d+1
```

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。

```
1  # coding=utf-8
2
3  l = [None] * 100
4
5  print type(l)                  # <type 'list'>
6  print type(type(l))            # <type 'type'>
7
8  print type(l) == type([])      # True
9  print type(l) == list          # True
10 print isinstance(l, list)      # True
```

Chapter 3

语句顺序

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
```

```
2     and c==d):
3     print 'all equal!'
```

3.2 三元表达式

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

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：

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

3.4 while 循环语句

```
1 while ... :
2     ...       # 执行的语句
3     ...       # 遇到 break 跳出整个循环
4     ...       # 遇到 continue 回到开头进行
5     ...       # pass 语句什么也不做，仅在语法上点一个位置
6 else :
7     ...       # 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

```
1 for ... in ... :
2     ...       # 执行的语句
3     ...       # 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
```

3.5.1 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 迭代器

3.6.1 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。

3.6.2 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
```

3.6.3 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)
sum([1,2,3,4,5])
any(['a','','c'])
all(['a','','c'])
```

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')
```

3.7 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”。

3.7.1 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：

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

```
4
5  print m[1]    # [4,5,6] print row 2
6  print m[1][2] # 6       print row 2, item 3
7
8  # show col 3 use list comprehension expression
9  print [row[1] for row in m]                    # [2,5,8]
10 print [row[1]+1 for row in m]                  # [3,6,9]
11 print [row[1] for row in m if row[1] % 2 == 0 ] # [2,8]
12
13 print [m[i][i] for i in [0,1,2]]          # [1,5,9]
14 print [c*2 for c in 'span']               # ['ss','pp','aa','mm']
```

Chapter 4

函数

4.1 函数定义

python 中，函数也是一个对象。def 语句定义一个函数对象，然后赋值给指定函数名变量：

```
def myAdd(a, b):
  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：

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

也可以把函数再赋值给另一个变量：

```
otherFunc = myadd
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。

函数参数列表支持name=value 形式的默认值：

```
1 def myAdd(a=1, b=2):
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。

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

4.2.1 参数传递

python 支持可变参数。当定义函数时，* 表示列表、** 表示字典：

```
1 def func(a, * pargs, ** kargs):
2     print a, pargs, kargs
3 
4 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：

```
1  lst = (1,2,3,4,5)
2  dic = {'a':1,'b':2,'c':3}
3  
4  func(*lst)                    # 1 (2, 3, 4, 5) {}
5  func(**dic)                   # 1 () {'c': 3, 'b': 2}
6  
7  def func(a,b,c):
8    print a
9  
10 func( *(1,2,3) )          # OK
11 func(  (1,2,3) )           # err
12 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 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'])
```

4.4 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):
    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
```

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  # coding=utf-8
2
3  class C1: pass             # Empty class
4                             # as an namespace object
5
6  a = C1()
7  a.name = "morgan"          # a.name is "morgan"
8
9  C1.name = "Class C1"       # C1.name is "C1"
10 b = C1()                   # b.name is "C1"
11                            # a.name still is "morgan"
12
```

```
13 def upperName(self):
14   print self.name.upper()
15
16 C1.showName = upperName     # add func to class
17 a.showName()
18 b.showName()
```

Listing 5.1: 类与实例

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。

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。类的成员只属于该类，不属于该类的实例。

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。

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。

实例对象的__class__ 属性记录了这个实例的类对象。

```
1  # coding=utf-8
2
3  class c1():
4    count = 0
5    def show(self):
6      print 'c1.show()'
7
8  a = c1()
9  b = c1()
10
11 print a.__class__           # class of instance
12
13 c1.count = 10               # c1.count is 10
14 a.count = 5                 # a.count is 5
15 b.count = 6                 # b.count is 6
```

```
16
17 a.show()
18 b.show()
```

Listing 5.2: 类与实例

5.3 类的属性与方法

python 中把数据保存在对象中，相关的操作（方法）在类中。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）。同样地也可以简章地通过赋值给变量增加一个成员。

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。

```
1  # coding=utf-8
2
3  class Employee():
4    def __init__(self, name):   # __init__ defing construter
5      self.name = name
6    def showName(self):
7      print self.name
8
9  morgan = Employee("Morgan")
10 morgan.showName()
11
12 morgan.showName = "new name"    # set value
13 morgan.nickName = "Jade"        # create new variable
14
15 print Employee.__dict__.keys()  # 大多数基于类的对象的命名空间字典
16 print morgan.__dict__.keys()    # 大多数基于类的对象的命名空间字典
```

Listing 5.3: 类的属性与方法

5.4 类的继承

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

类对象的成员__bases__ 是超类构成的元组

```
# coding=utf-8

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()'

print c3.__bases__              # 超类的元组

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

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'
# 对象成员赋值后不影响类成员的值
c1.myname                       # c1.myname still is 'aaa'
c3.myname                       # c3.myname still is 'aaa'
```

```
33
34 i1.show()                    # 调用重写的方法
35 i1.helloC1()                 # 调用到父类的方法
36 i1.helloC2()                 # 调用到父类的方法
```

Listing 5.4: 类继承

5.5 运算符重载

两头是下划线的方法名（__ 方法名 __）表示对运算符的重载。如：

__add__ 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。

```
1  # coding=utf-8
2
3  class C1():
4    def __init__(self, value):
5      self.data = value
6    def __add__(self, other):
7      return C1(self.data + other)
8    def __mul__(self, other):
9      return C1(self.data * other)
10
11 a = C1(5)
12 b = C1(10)
13
14 print (a + 3).data          # result is 8
15 print (b * 5).data          # result is 50
```

Listing 5.5: 运算符重载

Chapter 6

python 文档

6.1 PyDoc 查看文档内容

使用help() 函数查看帮助:

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

浏览 HTML 版的文档:

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

6.2 程序内文档

文件、类、函数头上的文档字符串__doc__ 会被自动封装为文档。

显示文档的方法:

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

Chapter 7

常用功能

7.1 输入输出

7.1.1 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
```

7.1.2 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')
```

7.1.3 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
```

7.2 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'

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