

# PYTHON学习笔记

BY William Wang

```
#!/usr/bin/env python      在Linux/Unix上可直接运行
# -*- coding: utf-8 -*-    指定编码
```

```
print('Hello world!')
```

```
name = input('please enter your name: ')
print('hello,', name)
```

## 字符串操作

如果字符串内部有很多换行，用\n写在一行里不好阅读，为了简化，Python允许用"""..."""的格式表示多行内容，可以自己试试：

```
print '''line1
... line2
... line3'''
line1
line2
line3
```

如果字符串里面有很多字符都需要转义，就需要加很多\，为了简化，Python还允许用r"表示"内部的字符串默认不转义，可以自己试试：

```
print '\t' \\
```

```
print r'\t' \\t\
```

## Boolean 值

```
True False
True and False

True or False
not True
```

## 空值 N

```
one
```

```
None
```

```
str.lower()
```

以Unicode表示的字符串用u'...'表示，比如：

```
print u'中文' 中文
```

```
u'中'
```

```
u'\u4e2d'
```

把u'xxx'转换为UTF-8编码的'xxx'用encode('utf-8')方法:

```
u'ABC'.encode('utf-8')
```

```
'BC'
```

```
u'中文'.encode('utf-8')
```

```
'\xe4\xb8\xad\xe6\x96\x87'
```

反过来, 把UTF-8编码表示的字符串'xxx'转换为Unicode字符串u'xxx'用decode('utf-8')方法:

```
'abc'.decode('utf-8')
```

```
u'abc'
```

```
'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
```

```
u'\u4e2d\u6587'
```

```
print '\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
```

```
中文
```

由于Python源代码也是一个文本文件, 所以, 当你的源代码中包含中文的时候, 在保存源代码时, 就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时, 为让它按UTF-8编码读取, 我们通常在文件开头写上这两行:

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

```
-- coding: utf-8 --
```

在Python中, 采用的格式化方式和C语言是一致的, 用%实现, 举例如下:

```
'Hello, %s' % 'world' 'ello, world'
```

```
'Hi, %s, you have $%d.' % ('Michael', 1000000) 'Hi, Michael, you have $1000000.'
```

有些时候, 字符串里面的%是一个普通字符怎么办? 这个时候就需要转义, 用%%来表一个%:

```
'growth rate: %d %%' % 7
```

```
'growth rate: 7 %'
```

在Python 3.x版本中, 把'xxx'和u'xxx'统一成Unicode编码, 即写不写前缀u都是一样的, 而以字节形式表示的字符串则必须加上b前缀: b'xxx'。

## list

```
classmates = ['Michael', 'Bob', 'Tracy'] len(classmates) classmates[0] classmates.append('Adam') classmates.insert(1, 'Jack') classmates.pop() ## 要删除list末尾的元素, 用pop()方法, 要删除指定位置的元素, 用pop(i)方法 classmates[1] = 'Sarah' L = ['Apple', 123, True]
```

```
L = [] cassmates[-1]
```

特殊的, [a]+[1,2,3]+[b]生成一个新的list

## tuple 元组:

---

tuple。tuple和list非常类似，但是tuple一旦初始化就不能修改 t=(1,2) t-1)

t1=(1,) ## 只有1个元素的tuple定义时必须加一个逗号,，来消除歧义

## dict

---

d = {'Michael': 95, 'Bob': 75, 'Tracy': 85} d['Michael'] d['Jack'] = 90 d['Jack'] = 88 'Thomas' in d d.pop('Bob') 和list比较，dict有以下几个特点：查找和插入的速度极快，不会随着key的增加而变慢；需要占用大量的内存，内存浪费多。而list相反：

查找和插入的时间随着元素的增加而增加；占用空间小，浪费内存很少。

## set

---

set和dict类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在set中，没有重复的key。要创建一个set，需要提供一个list作为输入集合：

```
s = set([1, 2, 3]) s { 2, 3}
```

```
s.add(1) s remove(1)
```

## for

---

for x in ...循环就是把每个元素代入变量x，然后执行缩进块的语句。

```
sm = 0
```

```
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]: sum = sum + x print(sum)
```

如果要计算1-100的整数之和，从1写到100有点困难，幸好Python提供一个range(函数，可以生成一个整数序列，再通过list()函数可以转换为list。比如range(5)生成的序列是从0开始小于5的整数：

```
list(range(5))
```

## while

---

```
sum = 0 n = 99 while n > 0: sum = sum + n
```

```
n = n - 2
```

```
print(sum)
```

## break&continue

```
while 1:
```

```
break
continue
```

while 结束时执行语句

```
while expression:
```

```
pass

else:
    pass
```

import math abs() max() min()

str() bool()

int() float()

## define a function

---

def my\_abs(x): if x >= 0: return x

```
else:
    return -x
```

pass

if 1: pass

## return a tuple

def return\_more\_than\_1():

```
    return 1,2,3,4,5
```

## 默认参数

必选参数在前，默认参数在后

```
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x

    return s
```

默认参数必须指向不变对象

## 可变参数

def calc(\*numbers): sum = 0

```
    for n in numbers:
        sum = sum + n * n

    return sum
```

\*nums表示把nums这个list的所有元素作为可变参数传进去。这种写法相当有用，而且很常

见。

##可变参数既可以直接传入：func(1, 2, 3)，又可以先组装list或tuple，再通过

args传入：func((1, 2, 3))：

## 关键字参数

```
extra = {'city': 'Beijing', 'job': 'Engineer'} person('Jack', 24, **extra) nme: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

**extra**表示把**extra**这个**dict**的所有**key-value**用关键字参数传入到函数的**kw**参数，**kw**将获得一个**dict**，注意**kw**获得的**dict**是**extra**的一份拷贝，对**kw**的改动不会影响到函数外的**extra**

要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收**city**和**job**作为关键字参数。这种方式定义的函数如下：

```
def person(name, age, *, city, job):
    print(name, age, city, job)
```

和关键字参数\***kw**不同，命名关键字参数需要一个特殊分隔符，\*后面的参数被视为命名关键字参数。调用方式如下：

```
person('Jack', 24, city='Beijing', job='Engineer') Jack 24 Beijing Engineer
```

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符\*了：

```
def person(name, age, *args, city, job):
    print(name, age, args, city, job)
```

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报

命名关键字参数可以有缺省值，从而简化调用： def person(name, age, \*, city='Beijing', job): print(name, age, city, job)

关键字参数既可以直接传入：func(a=1, b=2)，又可以先组装dict，再通过**kw**传入：**func({'a': 1, 'b': 2})**。

## 顺序

参数定义的顺序必须是：必选参数、默认参数、可变参数、命名关键字参数和关键字参数。比如：

```
def f1(a, b, c=0, *args, **kw): print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)
```

对于任意函数，都可以通过类似func(\*args \*\*kw)的形式调用它，无论它的参数是如何定义的

eg:

```
>
>> args = (1, 2, 3, 4)
>>> kw = {'d': 99, 'x': '## '}
>>> f1(*args, **kw)
>>> a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '## '}
>>> args = (1, 2, 3)
*args是可变参数，args接收的是一个tuple；
**kw是关键字参数，kw接收的是一个dict。
```

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（**stack**）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。

由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。

解决递归调用栈溢出的方法是通过尾递归优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

eg: 由于`return n * fact(n - 1)`引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

```
def fact(n): return fact_iter(n, 1)
```

```
def fact_iter(num, product): if num == 1: return product return fact_iter(num - 1, num * product)
```

针尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。

Python标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

## 切片（Slice）操作符

---

取前3个元素，用一行代码就可以完成切片：

```
>> L[0:3]

['Michael', 'Sarah', 'Tracy']
```

如果第一个索引是0，还可以省略：

```
L[3]
```

Python支持`L[-1]`取倒数第一个元素，那么它同样支持倒数切片，试试：

```
L[-2:]
```

```
['Bob', 'Jack']
```

```
L[-2:-1]
```

```
['Bob']
```

倒数第一个元素的索引是-1

什么都不写，只写`L[:]`就可以原样复制一个list：

```
L[:]
```

前10个数，每两个取一个：

```
L[10:2] [0, 2, 4, 6, 8]
```

tuple也是一种list，唯一区别是tuple不可变。因此，tuple也可以用切片操作，只操作的结果仍是tuple：

```
(0, 1, 2, 3, 4, 5)[3]
```

```
(0, 1, 2)
```

## 迭代

---

如果给定一个list或tuple，我们可以通过for循环来遍历这个list或tuple，这种遍历我们称为迭代（Iteration）。只要是可迭代对象，无论有无下标，都可以迭代，比如dict就可以迭代：

```
d = {'a': 1, 'b': 2, 'c': 3} for key in d: ... print(key) 因为dict的存储不是按照list的方式顺序排列，所以，迭代出的结果顺序很可能不一样。默认情况下，dict迭代的是key。如果要迭代value，可以用for value in d.values()，如果要同时迭代key和value，可以用for k, v in d.items()
```

如果要对list实现类似Java那样的下标循环怎么办？Python内置的enumerate函数可以把一个list变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：

```
for i, value in enumerate(['A', 'B', 'C']): ... print(i, value)
```

上面的for循环里，同时引用了两个变量，在Python里是很常见的，比如下面的代码

```
for x, y in [(1, 1), (2, 4), (3, 9)]: ... print(x, y)
```

## 列表生成式

```
[x * x for x in range(1, 11) if x % 2 == 0] [4, 16, 36, 64, 100]
```

还可以使用两层循环，可以生成全排列：

```
[m + n for m in 'ABC' for n in 'XYZ'] ['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

for循环其实可以同时使用两个甚至多个变量，比如dict的items()可以同时迭代key和value：

```
d = {'x': 'A', 'y': 'B', 'z': 'C'}
```

```
for k, v in d.items(): ... print(k, '=', v)
```

因此，列表生成式也可以使用两个变量来生成list：

```
d = {'x': 'A', 'y': 'B', 'z': 'C'} [k + '=' + v for k, v in d.items()]
```

## generator

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator。要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator：

eg: g = (x \* x for x in range(10))

可以通过next()函数获得generator的下一个返回值：

```
next(g)
```

A more widely used方法是使用for循环，因为generator也是可迭代对象：

```
g = (x * x for x in range(10)) for n in g: ... print(n) 赋值语句： a, b = b, a + b 相当于： t = (b, a + b) ## t是一个tuple  
a = t[0]
```

b = t[1] 但不必显式写出临时变量t就可以赋值。

### 定义generator的另一种方法

如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator： eg: def fib(max): n, a, b = 0, 0, 1 while n < max: yield b

```
a, b = b, a + b  
n = n + 1
```

```
return 'done'
```

变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行

## 迭代器

可以直接作用于for循环的对象统称为可迭代对象：Iterable。可以使用isinstance()判断一个对象是否是Iterable对象：凡是可作用于for循环的对象都是Iterable类型；凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；集合数据类型如list、dict、str、tuple等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。Python的for循环本质上就是通过不断调用next()函数实现的，例如：

为什么list、tuple、dict、str等数据类型不是Iterator？这是因为Python的Iterator对象表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回一个数据时它才会计算。

Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

## 函数('指针')调用

```
f = abs
```

```
f(-10) 10
```

## 高阶函数

一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。一最简单的高阶函数：

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

### 高阶函数 map()

map()函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。举例说明，比如我们有一个函数f(x)=x<sup>2</sup>，要把这个函数作用在一个list [1, 2, 3, 4, 5, 6, 7, 8, 9]上，就可以用map()实现 map(f,[1, 2, 3, 4, 5, 6, 7, 8, 9])

### 高阶函数 reduce()

reduce()函数接收两个参数，一个是函数，一个是Iterable,实现效果如下：

reduce(a,[1,2,3]) 等价于 a(a(1,2),3)

### 高阶函数 filter()

filter()接受一个函数和一个列表，之后保留列表中满足f(x)==True的项 用于实现筛法

### 高阶函数 sorted()

sorted(列表, key=比较函数, reverse=True反向排序)

ie. sorted() 原型为sorted(a是列表,key=默认比较函数如ASC2&数字大小, reverse=False)

### 闭包

函数作为返回值：

```
def lazy_sum(*args):
```



```
def sum():
    ax = 0
    for n in args:
        ax = ax + n
    return ax
return sum
```

当我们调用`lazy_sum()`时，返回的并不是求和结果，而是求和函数：

```
f = lazy_sum(1, 3, 5, 7, 9) f <function sum at 0x10452f668>
```

调用函数`f`时，才真正计算求和的结果：

```
>>> f()
25
```

这种称为“闭包（Closure）”的程序结构拥有极大的威力。

返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

```
>>> def count():
...     fs = []
...     for i in range(1, 4):
...         def f(j):
...             def g():
...                 return j*j
...             return g
...         fs.append(f(i))
...     return fs
```

亦即，返回函数不要直接使用循环变量的值，要使用另一层闭包结构作为中转：原理是返回函数只有在被调用时才会被执行。在循环中，每次`f()`被定义之后即被调用保证了每次传入`f()`的是当前循环变量的值。

也可以用私有变量实现

```
def count
():
    fs = []
    for i in range(1, 4):
        def f(j=i): ## 对于每个f(), j是私有的。
            return j*j

        fs.append(f)
    return fs
```

有点像C语言中的指针，定义的时候只是把这个`i`的指针（地址）传递进去了，只有函数在执行，才会去从具体的地址中把值读出来。但是这里`i`作为形参的时候也会把值取来，形参也相对于执行吧。

## 匿名函数lambda x: x \* x

lambda x:x\*x ## lambda para: return\_value flambda s:s+1 ## 赋值给函数

..... return lambda s:s+1

注意使用时函数体只有一条语句

## 装饰器

假设我们要增强函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改函数的定义，这种在代码运行期间动增加功能的方式，称之为“装饰器”（Decorator）本质上，decorator就是一个返回函数的高阶函数。所以，我们要定义一个能打印志的decorator，可以定义如下

```
def log(func): #函数 接受一个函数 生成一个新函数
    def wrapper(*args, **kw): ## 用一个外围函数拦截所有传入信息
        print 'call %s():' % func.__name__ ## 作为装饰的语句
        return func(*args, **kw) ## 将所拦截的信息传入原函数
    return wrapper ## 返回新生成的函数
```

可以这样调用：f=log(rint) print =log(print)

特别用法:

@log ## 在函数定义时使用 def now(): print '2013-12-25'

等价于now=log(now)

为返回的函数传入参数 形如func(para1)(para2) func(para1)产生一个函数func2 之后func2(para2) 籍此可以定义一个有传入的decorator

如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数，写出来会更复杂。比如，要自定义log的文本：

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

调用时如 log('qwert')(print)

在定义函数时调用

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

因为返回的那个wrapper()函数名字就是'wrapper'，所以，需要把原始函数的\_\_name\_\_等属性复制到wrapper()函数中，否则，有些依赖函数签名的代码执行就会出错。不需要编写wrapper.name = func.\_\_name\_\_这样的代码，Python内置的functools.wraps就是干这个事的，所以，一个完整的decorator的写法如下：

```
import functools

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

即在 `def wrapper(*args, **kw):` 之前 加入 `@functools.wraps(func)`

## 偏函数

---

固定一个函数的部分值 比如对`int('数字串',base=进制)`

方法一

```
def myint(a,base=2)
    return int(a,base)
```

方法二

```
import functools
int2 = functools.partial(int, base=2) ## 偏函数
```

偏函数的实质相当于： 当传入： `max2 = functools.partial(max, 10)` 实际上会把10作为\*args的一部分自动加到左边，也就是： `max2(5, 6, 7)` 相当于： `args = (10, 5, 6, 7)` `max(*args)`

## 内置函数

---

参见：<https://docs.python.org/2/library/functions.html>:

## Module & Package

---

eg:一个abc.py的文件就是一个名字叫abc的模块，一个xyz.py的文件就是一个名字叫xyz的模块

选择一个顶层包名，比如mycompany，按照如下目录存放：

```
mycompany
├── init.py
└── xyz.py
```

引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，`abc.py`模块的名字就变成了`mycompany.abc`，类似的，`xyz.py`的模块名变成了`mycompany.xyz`。

请注意，每一个包目录下面都会有一个`__init__.py`的文件，这个文件是必须存在的，否则，Python就把这个目录当成普通目录，而不是一个包。`init.py`可以是空文件，也可以有Python代码，因为`__init__.py`本身就是一个模块，而它的模块名就是`mycompany`。

可以有多个目录，组成多级层次的包结构。

## Using module

---

Python模块的标准文件模板：

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

' a test module '          #注释,文档注释也可以用特殊变量__doc__访问

__author__ = 'Michael Liao'
```

```
import sys
```

sys模块有一个argv变量，用list存储了命令行的所有参数。argv至少有一个元素，因为第一个参数永远是该.py文件的名称，例如：

运行python hello.py获得的sys.argv就是['hello.py']；

运行python hello.py Michael获得的sys.argv就是['hello.py', 'Michael']。

当我们在命令行运行hello模块文件时，Python解释器把一个特殊变量\_\_name\_\_置为\_\_main\_\_，而如果在其他地方导入该hello模块时，if判断将失败，因此，这种if测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

## 别名

导入模块时，还可以使用别名，这样，可以在运行时根据当前环境选择最合适的模块。比如Python标准库一般会提供StringIO和cStringIO两个库，这两个库的接口和功能是一样的，但是cStringIO是C写的，速度更快，所以，你会经常看到这样的写法：

```
try:
    import cStringIO as StringIO
except ImportError: # 导入失败会捕获到ImportError
    import StringIO
```

```
import ... as ...

try:
except xxxx:
```

## 作用域

类似xxx和\_xxx这样的函数或变量就是非公开的（private），不应该被直接引用，比如\_abc，\_\_abc等；

## 安装第三方模块

```
pip install 包名
```

常用：

- Python Imaging Library :PIL
- MySQL的驱动：MySQL-python，
- 用于科学计算的NumPy库：numpy，
- 用于生成文本的模板工具Jinja2

## sys.path

sys.path制定了import时寻找的目录

```
>>> import sys
>>> sys.path.append('/Users/michael/my_py_scripts')
```

设置环境变量PYTHONPATH也可规定import时寻找的目录

## 在python2中使用python3的功能

```
from __future__ import unicode_literals
```

# 面向对象编程

## 面向对象编程与面向过程编程

```
class Student(object): #class后面紧接着是类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该
                        #类是从哪个类继承下来的
```

#如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类

```
def __init__(self, name, score): #通过定义一个特殊的__init__方法，在创建实例的时候，就把name, score
等属性绑上去
```

```
# self不需要传，Python解释器自己会把实例变量传进去
```

```
self.name = name
```

```
self.__score = score #实例的变量名如果以__开头，就变成了一个私有变量（private），只有内部可以访
问，外部不能访问
```

```
#可以用默认参数、可变参数和关键字参数。
```

#需要注意的是，在Python中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量

```
def print_score(self):
    print '%s: %s' % (self.name, self.score)
```

#给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。面向对象的程序写出来就像这样：

```
bart = Student('Bart Simpson', 59) #创建实例是通过类名+()实现的
lisa = Student('Lisa Simpson', 87)
bart.print_score()
lisa.print_score()
```

简单地说，Class is the model of Instance

ps:双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问\_\_name是因为Python解释器对外把\_\_name变量改成了\_Student\_\_name，所以，仍然可以通过\_Student\_\_name来访问\_\_name变量。但是：不同版本的Python解释器可能会把\_\_name改成不同的变量名。