# python 速成指南

作者：doudehou@gmail.com

## 第一节. 过程式 python

python 的一个特点是不通过大括号 {} 来划定代码块，而是通过缩进。如果和 C/C++ 类比的话，就是在左括号的地方不要换行，然后用一个冒号 (:) 替代， C/C++ 大括号内部的东西，缩进一个 tab 或者几个空格都可以（但需要保持一致），比如：

if (x < 2):

print 'x < 2'

elif (x > 2):

print 'x > 2'

else:

print 'bingo!'

print 'x = 2'

注意两点：一是 python 语句结尾处没有分号(;)作为结束标记。二是和 C/C++ 不同，没有 else if，而是用 elif 替代，相当于可以少打几个字符吧。

### 类型系统

比如 int，string 等，type() 可以返回数据的类型，如：

>>> type(1)

<type 'int'>

>>> type('123')

<type 'str'>

python 尽管在声明变量的时候不指定类型，但变量其实是有类型的，用 c++0x 的概念来表达的话，实际上 python 的变量好像都是 auto 的，类型自动根据赋值推导出来。所以这样：

name = 'ddh'

verb = ' is '

noun = ' good man'

sentence = name + verb + noun

没有问题，但：

name = 'cyberscorpio'

age = 32

sentence = name + age

就会引发异常，因为字符串和数字不能直接相加。

可以通过 int() 或者 str() 强转类型，如上一句改成： sentence = name + str(age) 就不会有问题了。

### 容器类型

python 提供好用的两个容器：list 和 dict。插句题外话，其实最好用的容器还是 PHP 提供的关联数组，一个数组就包括了 python 中 list 和 dict 的全部功能，实在是很赞。

#### list

类似 array 的概念，例如：

lst = list()

lst.append('123')

lst.append('456')

lst.append(1000)

print lst

for x in lst:

print x, ' type is ', type(x)

注意 list 内的数据可以是不同类型的，这一点会很方便。上面的输出是：

>>> print lst

['123', '456', 1000]  **# 注意这里是 list 的字面表示方法，如 lst = ['123', '456', 1000]，lst 自动成为一个 list**

>>> for x in lst:

... print x, ' type is ', type(x)

...

123 type is <type 'str'>

456 type is <type 'str'>

1000 type is <type 'int'> # 这个是整型的数据

#### dict

类似于 std::map 的概念，当然，和 list 类似，dict 的 key 和 value 不要求是同一种类型。如：

dct = dict()

dct['name'] = 'cyberscorpio'

dct['age'] = 32

dct['sex'] = 'male'

print dct

for k in dct:

print k, ' is ', dct[k]

输出为：

>>> print dct

{'age': 32, 'name': 'cyberscorpio', 'sex': 'male'} **# 注意这里是 dict 的字面表示方法，如 dct = {'age':32, 'name':'cyberscorpio', 'sex':'male'}，则 dct 自动成为一个 dict**

>>> for k in dct:

... print k, ' is ', dct[k]

...

age is 32

name is cyberscorpio

sex is male

判断一个 key 是否在 dict 中：

dct = {'name' : 'ddh', 'age' : 32, }

if 'name' in dct:

print 'the dict has a name and we will delete it...'

del dct['name']

可以使用 del 删除这个 key。另外 if key not in dict 可以判断这个 key 是否 “不在” dict 中。

list 和 dict 均为某种对象，它们支持的方法 （method） 可以通过语言内置的 dir() 来获取，比如：

>>> dir(lst)

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_delslice\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getslice\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_setslice\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

比如 append()，insert()，sort() 什么的，都是 list 很常用的方法。而：

>>> dir(dct)

['\_\_class\_\_', '\_\_cmp\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'clear', 'copy', 'fromkeys', 'get', 'has\_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']

has\_key()，keys()，values() 等都是 dict 常用的方法。

用 dir('123') 则可以看到 string 支持的所有方法，总之还是比较方便的。

### 循环

简单的 for 循环，如：

fox x in 容器:

对 x 做操作

这是对容器进行枚举的很方便的做法。

其他的循环方式还有 while，如：

while <测试>:

循环体

注意在循环中我们仍然有 **break**, **continue** 等跳出或者继续循环的指令，和 C/C++ 是一致的。

### 函数

函数定义：

def 函数名 (参数列表):

函数内容

值得注意的是，函数中变量默认是 local 的，如果要访问全局变量，那么需要在函数中声明其为 **global** 的，如：

bar = 0

def foo ():

global bar

bar = bar + 1

print str(bar)

**参数列表**这里面颇有玄机，python 具备和 C++ 相同的两种参数列表结构，就是 (var1, var2) 和 (var1, var2=default value) 这两种，但除此之外呢，和 C/C++ 的 … 类似，python 还有两种针对不定参数个数的独门武功，举例说明：

def foo (\*params):

if len(params) == 0:

print 'no parameter' # 如果调用方式为 foo() 则进入这里

else

print params # 如果调用方式为 foo(1, 2, 3, 4)，则这里输出为 (1, 2, 3, 4)，params 是一个 tuple

def bar (\*\*params):

if len(params) == 0:

print 'no parameter' # bar()

else

print params # bar(name='ddh', age=32)，则这里输出为 {'name' : 'ddh', 'age' : 32}，params 是一个 dict。

对于 \*\* 这种方式，调用函数的时候，必须指定参数的名称和值，名称不需要带引号，但进入函数以后，自然成为同名的字符串，如上可见。

Python 所有的变量（不仅仅是传参）都是基于引用的，但是 python 的对象分为两种，一种叫 immutable，还有一种叫 mutable，区别在哪里呢？后者具有方法来改变自己，比如 list.append() 等，而 str 就是 immutable 的，也即，所有的 str 方法都不能改变自己（比如 str.replace() 是返回一个被替换过的 str，原来的 str 不变）。这样的话，我们这么来看：

def foo (s):

s = '456'

name = '123'

foo(name)

首先，name 本身不是对象，它是对一个 str 且内容为 '123' 的 str 对象的引用。其次，在 foo 函数中， s 是对同一个对象的，引用，而不是对 name 的引用。那么可以断定：

1. 对 s 的改变无法影响到 name，因为 name 并未被引用进来。
2. s = '456' 只是把 s 换成了对另一个内容为 '456' 的 str 对象的引用，无法改变之前那个 '123' 的 str 对象。
3. 因为 str 不提供改变自己的方法，所以，foo 函数无法改变 name 所指向的对象的值。

这么来看的话，immutable 对象创建出来之后就不能变了，比如：

s = '123'

s = '456'

这里第一句，s 指向一个内容为 '123' 的 str 对象，第二句，s 指向一个内容为 '456' 的 str 对象。第一句中的对象，没有被任何人引用了，之后会被 “废料收集” 程序废掉，但我们不能改变它的内容。另：参见附录一 《对 python 中引用问题的再思考》。

除此之外，函数就没什么神奇的地方了，比如可以通过 return 返回一个值等等，这些都是老僧禅痰了。

### TRY...EXCEPT

对于出现 traceback 的错误来讲，可以通过 try...except 来挽救。比如：

def safeint(sval):

try:

return int(sval)

except:

return -1

这样，safeint('xxxx') 就不会出现 traceback 了，因为 int('xxx') 会被 except 捕获，正常返回 -1。

## 第二节. 面向对象的 python

其实前面的 str, list 还有 dict 都是某种意义上的类 (class)，只不过是内置的而已。如前所述，dir() 可以列出类和对象的方法 (method)。

### class 和 self

和 c++ 一样，python 也用 class 定义一个类：

class 类名称:

"类的文档字串，用来简单说明这个类"

成员变量

def \_\_int\_\_ (self, 参数): # [可选] 类似构造函数，但 \_\_init\_\_ 被(自动)调用时，对象已然被创建出来了

函数体

def \_\_del\_\_ (self): # [可选] 类似析构

函数体

def 成员函数 (self, 参数):

函数体

习惯上，和 c++ 不同的是，python 一般使用 self （是一种习惯而非强制，在 def 的时候可以使用其他的名字比如 this）而不是 this，但两者功能上类似，值得注意的是，self **需要在方法的参数中写定**，并且引用成员变量的时候，一定要通过 self。比如：

class Person:

"store information of a person"

name = ''

age = 0

sex = 'male'

def textOut(self): # 这个 self 不可少

print 'name is: ', self.name, ' age is ', self.age, ' and sex is ', self.sex # 对成员变量的引用，亦**必须通过 self**

@staticmethod

def sayHello ():

print 'hello,world!'

p = Person()

p.textOut()

p.sayHello()

Person.sayHello()

如果成员函数不带 self 参数，则类似 c++ 中静态函数的概念，不能访问成员变量，但是需要在函数前面加 @staticmethod 修饰 (独占一行)，静态方法可以通过对象调用，亦可以通过类名调用。如上。

另外需要注意的是很多以两个下划线开头和结尾的函数（专用方法），这些往往都有特殊用途，比如上面提到的 \_\_init\_\_ 和 \_\_del\_\_，还有 \_\_add\_\_, \_\_eq\_\_, \_\_ge\_\_, \_\_gt\_\_, \_\_lt\_\_, \_\_le\_\_, \_\_ne\_\_, \_\_getitem\_\_, \_\_str\_\_ 等等。比如 \_\_getitem\_\_ 用来实现 [] 操作，而 \_\_str\_\_ 用来实现 str() 操作，等等。

### 继承

python 通过这样的语法形式来实现继承：

class Derived (Base1[, Base2[, …]]]):

类实现

可以有多个逗号分割开的基类，用以实现多继承。有两个细节要注意：

如果定义了 \_\_init\_\_ 函数，那么必须在 \_\_init\_\_ 中调用基类的 \_\_init\_\_，方法为 基类名.\_\_init\_\_(self[, 其他参数])

调用基类的函数，务必手工添加 self 参数

一个例子：

class Namable:

name = ''

def \_\_init\_\_ (self, name):

self.name = name

def output (self):

print 'name is:', self.name

class Agable:

age = 0

def \_\_init\_\_ (self, age):

self.age = age

def output (self):

print 'age is:', self.age

class Person (Namable, Agable):

sex = 'male'

def \_\_init\_\_ (self, name, age, sex = 'male'):

Namable.\_\_init\_\_(self, name) **# 显式调用基类的 \_\_init\_\_，带 self 参数**

Agable.\_\_init\_\_(self, age)

self.sex = sex

def output (self):

Namable.output(self)  **# 调用基类的函数，亦需要带上 self 参数**

Agable.output(self)

print 'sex is:', self.sex

def do\_output (self):

self.output() **# 非基类函数，则无需手工加上 self 参数**

p = Person('doudehou', 32)

p.do\_output()

## 附录

### 附录一. 关于 python 中引用问题的再思考

python 中所有的变量都是引用，这一点需要时刻牢记在心。什么意思呢？举个例子， d1 = dict() ，这句话的意思就是，我们首先创建了一个类型为 dict 的 （匿名）对象，接着创建了一个变量 d1，然后让 d1 “引用” 这个 dict 的对象，如果类比于 C 语言的话，可以理解为 dict \*d1 = new dict()。d1 它只是一个引用。

这里和 C 语言存在一个巨大的差异。这个差异是什么呢？我们知道 C 语言提供了解引用的操作符 (\*)，这样程序员可以通过 \*d1 来获得这个对象的本体，而作为脚本语言的 python，却并没有（也没有必要）提供这个功能。因此，除了通过 “引用” （换句话说，就是指针）的方式之外，我们无从触摸到这个对象的本体。

想象一下，如果 C 语言中没有了对指针做 \* 的运算，我们要如何来修改某个对象呢？比如：

char c = 'A';

char \*p = &c;

我们手上只有一个 p，要怎么修改 \*p 的内容呢？毫无疑问，我们没有办法。而这就正是 python 里的现实情况。char 就是属于 immutable 的类型，我们再举一个 C++ 的例子：

class foo {

char m\_c;

public:

foo () : m\_c('A') {}

void set (char c) {

m\_c = c;

}

char get () {

return m\_c;

}

};

foo \*p = new foo();

std::cout << "foo: " << p->get() << std::endl;

p->set('B');

std::cout << "foo: " << p->get() << std::endl;

在上面这个例子里，如果说我们仍然没有对指针的 \* 运算，但我们依然可以改变 \*p 的内容，这是因为 p 所指向的对象，具有修改自身的能力（提供了相关的方法）。这种类型，在 python 里面，就是 mutable 的。我们常见的 list，dict 等等，就属于此类。而 python 的字符串类，则属于 immutable 的。

这里有两个地方需要注意：

1. 变量赋值，实际上并没有产生对象的拷贝，仅仅是 “引用” （指针） 的拷贝。比如：

d1 = {'1' : 1, '2' : 2, }

d2 = d1

d2['3'] = 3

for k in d1:

print k, ' => ', d1[k]

上面的代码，我们会发现，d1['3'] 也存在了，而且值就是 3。所以我们可以把 python 里面的所有变量都想象成指针；所有的变量对变量的 = 操作都想象成指针的赋值，所有的 . 操作都想象成 ->，那么则庶几不会有问题了。

1. 通过显式构造函数的方法来拷贝对象，还是上面的例子，如果我们用：

d1 = {'1' : 1, '2' : 2, }

d2 = dict(d1)

d2['3'] = 3

for k in d1:

print k, ' => ', d1[k]

这样，d2 和 d1 就没有一毛钱的关系了，这种情况下，dict(d1) 产生了一个新的（匿名）dict 对象，同时将 d2 变成对它的引用。如果我们把所有 x = Type(var) 的赋值想象成 x = new Type(var) 就相当于抓住了事物的本质。