

目錄

课程介绍	0
面向对象进阶	1
元类	1.1
python是动态语言	1.2
__slots__	1.3
生成器	1.4
迭代器	1.5
闭包	1.6
装饰器	1.7
其他的知识点	2
import导入模块	2.1
循环导入	2.2
作用域	2.3
==、is	2.4
深拷贝、浅拷贝	2.5
进制、位运算	2.6
私有化	2.7
属性property	2.8
其他的知识点	3
垃圾回收(一)	3.1
垃圾回收(二)	3.2
垃圾回收(三)-gc模块	3.3
内建属性	3.4
内建函数	3.5

集合set	3.6
functools	3.7
模块	3.8
调试	3.9
编码风格	3.10

元类

1. 类也是对象

在大多数编程语言中，类就是一组用来描述如何生成一个对象的代码段。在Python中这一点仍然成立：

```
>>> class ObjectCreator(object):
...     pass
...
>>> my_object = ObjectCreator()
>>> print my_object
<__main__.ObjectCreator object at 0x8974f2c>
```

但是，Python中的类远不止如此。类同样也是一种对象。是的，没错，就是对象。只要你使用关键字class，Python解释器在执行的时候就会创建一个对象。

下面的代码段：

```
>>> class ObjectCreator(object):
...     pass
...
```

将在内存中创建一个对象，名字就是ObjectCreator。这个对象（类对象ObjectCreator）拥有创建对象（实例对象）的能力。但是，它的本质仍然是一个对象，于是乎你可以对它做如下的操作：

1. 你可以将它赋值给一个变量
2. 你可以拷贝它
3. 你可以为它增加属性
4. 你可以将它作为函数参数进行传递

下面是示例：

```
>>> print ObjectCreator      # 你可以打印一个类，因为它其实也是一个对象
<class '__main__.ObjectCreator'>
>>> def echo(o):
...     print o
...
>>> echo(ObjectCreator)          # 你可以将类做为参数传给函数
<class '__main__.ObjectCreator'>
>>> print hasattr(ObjectCreator, 'new_attribute')
False
>>> ObjectCreator.new_attribute = 'foo' # 你可以为类增加属性
>>> print hasattr(ObjectCreator, 'new_attribute')
True
>>> print ObjectCreator.new_attribute
foo
>>> ObjectCreatorMirror = ObjectCreator # 你可以将类赋值给一个变量
>>> print ObjectCreatorMirror()
<__main__.ObjectCreator object at 0x8997b4c>
```

2. 动态地创建类

因为类也是对象，你可以在运行时动态的创建它们，就像其他任何对象一样。首先，你可以在函数中创建类，使用class关键字即可。

```
>>> def choose_class(name):
...     if name == 'foo':
...         class Foo(object):
...             pass
...     return Foo      # 返回的是类，不是类的实例
... else:
...     class Bar(object):
...         pass
...     return Bar
```

```

...
>>> MyClass = choose_class('foo')
>>> print MyClass           # 函数返回的是类，不是类的实例
<class '__main__.Foo'>
>>> print MyClass()        # 你可以通过这个类创建类实例，也就是对象
<__main__.Foo object at 0x89c6d4c>

```

但这还不够动态，因为你仍然需要自己编写整个类的代码。由于类也是对象，所以它们必须是通过什么东西来生成的才对。**当你使用class关键字时，Python解释器自动创建这个对象。**但就和Python中的大多数事情一样，Python仍然提供给你手动处理的方法。

还记得内建函数type吗？这个古老但强大的函数能够让你知道一个对象的类型是什么，就像这样：

```

>>> print type(1) #数值的类型
<type 'int'>
>>> print type("1") #字符串的类型
<type 'str'>
>>> print type(ObjectCreator()) #实例对象的类型
<class '__main__.ObjectCreator'>
>>> print type(ObjectCreator) #类的类型
<type 'type'>

```

仔细观察上面的运行结果，发现使用type对ObjectCreator查看类型是，答案为type，是不是有些惊讶。。。看下面

3. 使用type创建类

type还有一种完全不同的功能，动态的创建类。

type可以接受一个类的描述作为参数，然后返回一个类。（要知道，根据传入参数的不同，同一个函数拥有两种完全不同的用法是一件很傻的事情，但这在Python中是为了保持向后兼容性）

type可以像这样工作：

type(类名, 由父类名称组成的元组 (针对继承的情况, 可以为空) , 包含属性的字典 (名称和值))

比如下面的代码：

```
In [2]: class Test: #定义了一个Test类
...:     pass
...:
In [3]: Test() #创建了一个Test类的实例对象
Out[3]: <__main__.Test at 0x10d3f8438>
```

可以手动像这样创建：

```
Test2 = type("Test2", (), {}) #定了一个Test2类
In [5]: Test2() #创建了一个Test2类的实例对象
Out[5]: <__main__.Test2 at 0x10d406b38>
```

我们使用"Test2"作为类名，并且也可以把它当做一个变量来作为类的引用。类和变量是不同的，这里没有任何理由把事情弄的复杂。即type函数中第1个实参，也可以叫做其他的名字，这个名字表示类的名字

```
In [23]: MyDogClass = type('MyDog', (), {})
In [24]: print MyDogClass
<class '__main__.MyDog'>
```

使用help来测试这2个类

```
In [10]: help(Test) #用help查看Test类
Help on class Test in module __main__:
```

```
class Test(builtins.object)
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)
```

In [8]: help(Test2) #用help查看Test2类

Help on class Test2 in module __main__:

```
class Test2(builtins.object)
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)
```

4. 使用type创建带有属性的类

type 接受一个字典来为类定义属性，因此

```
>>> Foo = type('Foo', (), {'bar':True})
```

可以翻译为：

```
>>> class Foo(object):
...     bar = True
```

并且可以将Foo当成一个普通的类一样使用：

```
>>> print Foo
<class '__main__.Foo'>
>>> print Foo.bar
True
>>> f = Foo()
>>> print f
<__main__.Foo object at 0x8a9b84c>
>>> print f.bar
True
```

当然，你可以向这个类继承，所以，如下的代码：

```
>>> class FooChild(Foo):
...     pass
```

就可以写成：

```
>>> FooChild = type('FooChild', (Foo,), {})
>>> print FooChild
<class '__main__.FooChild'>
>>> print FooChild.bar    # bar属性是由Foo继承而来
True
```

注意：

- type的第2个参数，元组中是父类的名字，而不是字符串
- 添加的属性是类属性，并不是实例属性

5. 使用type创建带有方法的类

最终你会希望为你的类增加方法。只需要定义一个有着恰当签名的函数并将其实作为属性赋值就可以了。

添加实例方法

```
In [46]: def echo_bar(self): #定义了一个普通的函数
....:     print(self.bar)
....:

In [47]: FooChild = type('FooChild', (Foo,), {'echo_bar': echo_ba
In [48]: hasattr(Foo, 'echo_bar') #判断Foo类中，是否有echo_bar这个属性
Out[48]: False

In [49]: 

In [49]: hasattr(FooChild, 'echo_bar') #判断FooChild类中，是否有echo_bar这个属性
Out[49]: True

In [50]: my_foo = FooChild()

In [51]: my_foo.echo_bar()
True
```

添加静态方法

```
In [36]: @staticmethod
....: def testStatic():
....:     print("static method ....")
....:

In [37]: Foochild = type('Foochild', (Foo,), {"echo_bar":echo_bar,
....: testStatic})

In [38]: fooclid = Foochild()

In [39]: fooclid.testStatic
Out[39]: <function __main__.testStatic>

In [40]: fooclid.testStatic()
static method ....
```

```
In [41]: fooclid.echo_bar()  
True
```

添加类方法

```
In [42]: @classmethod  
....: def testClass(cls):  
....:     print(cls.bar)  
....:
```

```
In [43]:
```

```
In [43]: Foochild = type('Foochild', (Foo,), {"echo_bar":echo_bar,  
....: testStatic, "testClass":testClass})
```

```
In [44]:
```

```
In [44]: fooclid = Foochild()
```

```
In [45]: fooclid.testClass()
```

```
True
```

你可以看到，在Python中，类也是对象，你可以动态的创建类。这就是当你使用关键字class时Python在幕后做的事情，而这就是通过元类来实现的。

6. 到底什么是元类（终于到主题了）

元类就是用来创建类的“东西”。你创建类就是为了创建类的实例对象，不是吗？但是我们已经学习到了Python中的类也是对象。

元类就是用来创建这些类（对象）的，元类就是类的类，你可以这样理解为：

```
MyClass = MetaClass() #使用元类创建出一个对象，这个对象称为“类”
MyObject = MyClass() #使用“类”来创建出实例对象
```

你已经看到了type可以让你像这样做：

```
MyClass = type('MyClass', (), {})
```

这是因为函数type实际上是一个元类。type就是Python在背后用来创建所有类的元类。现在你想知道那为什么type会全部采用小写形式而不是Type呢？好吧，我猜这是为了和str保持一致性，str是用来创建字符串对象的类，而int是用来创建整数对象的类。type就是创建类对象的类。你可以通过检查

`__class__`属性来看到这一点。**Python中所有的东西，注意，我是指所有的东西——都是对象。这包括整数、字符串、函数以及类。**它们全部都是对象，而且它们都是从一个类创建而来，这个类就是type。

```
>>> age = 35
>>> age.__class__
<type 'int'>
>>> name = 'bob'
>>> name.__class__
<type 'str'>
>>> def foo(): pass
>>> foo.__class__
<type 'function'>
>>> class Bar(object): pass
>>> b = Bar()
>>> b.__class__
<class '__main__.Bar'>
```

现在，对于任何一个`__class__`的`__class__`属性又是什么呢？

```
>>> a.__class__.__class__
<type 'type'>
>>> age.__class__.__class__
```

```
<type 'type'>
>>> foo.__class__.__class__
<type 'type'>
>>> b.__class__.__class__
<type 'type'>
```

因此，元类就是创建类这种对象的东西。type就是Python的内建元类，当然了，你也可以创建自己的元类。

7. __metaclass__属性

你可以在定义一个类的时候为其添加__metaclass__属性。

```
class Foo(object):
    __metaclass__ = something...
    ...省略...
```

如果你这么做了，Python就会用元类来创建类Foo。小心点，这里面有些技巧。你首先写下class Foo(object)，但是类Foo还没有在内存中创建。Python会在类的定义中寻找__metaclass__属性，如果找到了，Python就会用它来创建类Foo，如果没有找到，就会用内建的type来创建这个类。把下面这段话反复读几次。当你写如下代码时：

```
class Foo(Bar):
    pass
```

Python做了如下的操作：

1. Foo中有__metaclass__这个属性吗？如果是，Python会通过__metaclass__创建一个名字为Foo的类(对象)
2. 如果Python没有找到__metaclass__，它会继续在Bar（父类）中寻找__metaclass__属性，并尝试做和前面同样的操作。
3. 如果Python在任何父类中都找不到__metaclass__，它就会在模块层次中去寻找__metaclass__，并尝试做同样的操作。

4. 如果还是找不到`__metaclass__`, Python就会用内置的`type`来创建这个类对象。

现在的问题就是，你可以在`__metaclass__`中放置些什么代码呢？答案就是：可以创建一个类的东西。那么什么可以用来创建一个类呢？`type`，或者任何使用到`type`或者子类化`type`的东东都可以。

8. 自定义元类

元类的主要目的就是为了当创建类时能够自动地改变类。通常，你会为API做这样的事情，你希望可以创建符合当前上下文的类。

假想一个很傻的例子，你决定在你的模块里所有的类的属性都应该是大写形式。有好几种方法可以办到，但其中一种就是通过在模块级别设定`__metaclass__`。采用这种方法，这个模块中的所有类都会通过这个元类来创建，我们只需要告诉元类把所有的属性都改成大写形式就万事大吉了。

幸运的是，`__metaclass__`实际上可以被任意调用，它并不需要是一个正式的类。所以，我们这里就先以一个简单的函数作为例子开始。

python2中

```
#-*- coding:utf-8 -*-
def upper_attr(future_class_name, future_class_parents, future_c

#遍历属性字典，把不是__开头的属性名字变为大写
newAttr = {}
for name,value in future_class_attr.items():
    if not name.startswith("__"):
        newAttr[name.upper()] = value

#调用type来创建一个类
return type(future_class_name, future_class_parents, newAttr

class Foo(object):
    __metaclass__ = upper_attr #设置Foo类的元类为upper_attr
    bar = 'bip'
```

```
print(hasattr(Foo, 'bar'))
print(hasattr(Foo, 'BAR'))

f = Foo()
print(f.BAR)
```

python3中

```
#-*- coding:utf-8 -*-
def upper_attr(future_class_name, future_class_parents, future_c

#遍历属性字典，把不是__开头的属性名字变为大写
newAttr = {}
for name,value in future_class_attr.items():
    if not name.startswith("__"):
        newAttr[name.upper()] = value

#调用type来创建一个类
return type(future_class_name, future_class_parents, newAttr

class Foo(object, metaclass=upper_attr):
    bar = 'bip'

print(hasattr(Foo, 'bar'))
print(hasattr(Foo, 'BAR'))

f = Foo()
print(f.BAR)
```

现在让我们再做一次，这一次用一个真正的class来当做元类。

```
#coding=utf-8

class UpperAttrMetaClass(type):
```

```
# __new__ 是在__init__之前被调用的特殊方法
# __new__是用来创建对象并返回之的方法
# 而__init__只是用来将传入的参数初始化给对象
# 你很少用到__new__，除非你希望能够控制对象的创建
# 这里，创建的对象是类，我们希望能够自定义它，所以我们这里改写__new__
# 如果你希望的话，你也可以在__init__中做些事情
# 还有一些高级的用法会涉及到改写__call__特殊方法，但是我们这里不用
def __new__(cls, future_class_name, future_class_parents, fu
    #遍历属性字典，把不是__开头的属性名字变为大写
    newAttr = {}
    for name,value in future_class_attr.items():
        if not name.startswith("__"):
            newAttr[name.upper()] = value

    # 方法1：通过'type'来做类对象的创建
    # return type(future_class_name, future_class_parents, r

    # 方法2：复用type.__new__方法
    # 这就是基本的OOP编程，没什么魔法
    # return type.__new__(cls, future_class_name, future_cla

    # 方法3：使用super方法
    return super(UpperAttrMetaClass, cls).__new__(cls, futur

#python2的用法
class Foo(object):
    __metaclass__ = UpperAttrMetaClass
    bar = 'bip'

# python3的用法
# class Foo(object, metaclass = UpperAttrMetaClass):
#     bar = 'bip'

    print(hasattr(Foo, 'bar'))
    # 输出: False
    print(hasattr(Foo, 'BAR'))
    # 输出:True
```

```
f = Foo()  
print(f.BAR)  
# 输出:'bip'
```

就是这样，除此之外，关于元类真的没有别的可说的了。但就元类本身而言，它们其实是很简单的：

1. 拦截类的创建
2. 修改类
3. 返回修改之后的类

究竟为什么要使用元类？

现在回到我们的大主题上来，究竟是为什么你会去使用这样一种容易出错且晦涩的特性？好吧，一般来说，你根本就用不上它：

“元类就是深度的魔法，99%的用户应该根本不此操心。如果你想搞清楚究竟是否需要用到元类，那么你就不需要它。那些实际用到元类的人都非常清楚地知道他们需要做什么，而且根本不需要解释为什么要用元类。”——

Python界的领袖 Tim Peters

python是动态语言

1. 动态语言的定义

动态编程语言 是 高级程序设计语言 的一个类别，在计算机科学领域已被广泛应用。它是一类 在 运行时可以改变其结构 的语言：例如新的函数、对象、甚至代码可以被引进，已有的函数可以被删除或是其他结构上的变化。动态语言目前非常具有活力。例如JavaScript便是一个动态语言，除此之外如 PHP 、 Ruby 、 Python 等也都属于动态语言，而 C 、 C++ 等语言则不属于动态语言。----来自 维基百科

2. 运行的过程中给对象绑定(添加)属性

```
>>> class Person(object):
    def __init__(self, name = None, age = None):
        self.name = name
        self.age = age

>>> P = Person("小明", "24")
>>>
```

在这里，我们定义了1个类Person，在这个类里，定义了两个初始属性name和age，但是人还有性别啊！如果这个类不是你写的是不是你会尝试访问性别这个属性呢？

```
>>> P.sex = "male"
>>> P.sex
'male'
>>>
```

这时候就发现问题了，我们定义的类里面没有sex这个属性啊！怎么回事呢？这就是动态语言的魅力和坑！这里实际上就是 动态给实例**绑定属性**！

3. 运行的过程中给类绑定(添加)属性

```
>>> P1 = Person("小丽", "25")
>>> P1.sex

Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    P1.sex
AttributeError: Person instance has no attribute 'sex'
>>>
```

我们尝试打印P1.sex，发现报错，P1没有sex这个属性！---- 给P这个实例绑定属性对P1这个实例不起作用！那我们要给所有的Person的实例加上sex属性怎么办呢？答案就是直接给Person绑定属性！

```
>>> Person.sex = None #给类Person添加一个属性
>>> P1 = Person("小丽", "25")
>>> print(P1.sex) #如果P1这个实例对象中没有sex属性的话，那么就会访问它的
None #可以看到没有出现异常
>>>
```

4. 运行的过程中给类绑定(添加)方法

我们直接给Person绑定sex这个属性，重新实例化P1后，P1就有sex这个属性了！那么function呢？怎么绑定？

```
>>> class Person(object):
    def __init__(self, name = None, age = None):
        self.name = name
        self.age = age
```

```
def eat(self):
    print("eat food")

>>> def run(self, speed):
    print("%s在移动，速度是 %d km/h"%(self.name, speed))

>>> P = Person("老王", 24)
>>> P.eat()
eat food
>>>
>>> P.run()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    P.run()
AttributeError: Person instance has no attribute 'run'

>>>
>>>
>>> import types
>>> P.run = types.MethodType(run, P)
>>> P.run(180)
老王在移动，速度是 180 km/h
```

既然给类添加方法，是使用 `类名.方法名 = xxxx`，那么给对象添加一个方法也是类似的 `对象.方法名 = xxxx`

完整的代码如下：

```
import types

#定义了一个类
class Person(object):
    num = 0
    def __init__(self, name = None, age = None):
```

```
        self.name = name
        self.age = age
    def eat(self):
        print("eat food")

#定义一个实例方法
def run(self, speed):
    print("%s在移动, 速度是 %d km/h"%(self.name, speed))

#定义一个类方法
@classmethod
def testClass(cls):
    cls.num = 100

#定义一个静态方法
@staticmethod
def testStatic():
    print("---static method----")

#创建一个实例对象
P = Person("老王", 24)
#调用在class中的方法
P.eat()

#给这个对象添加实例方法
P.run = types.MethodType(run, P)
#调用实例方法
P.run(180)

#给Person类绑定类方法
Person.testClass = testClass
#调用类方法
print(Person.num)
Person.testClass()
print(Person.num)

#给Person类绑定静态方法
Person.testStatic = testStatic
```

```
#调用静态方法  
Person.testStatic()
```

5. 运行的过程中删除属性、方法

删除的方法：

1. del 对象.属性名
2. delattr(对象, "属性名")

通过以上例子可以得出一个结论：相对于动态语言，静态语言具有严谨性！
所以，玩动态语言的时候，小心动态的坑！

那么怎么避免这种情况呢？请使用__slots__，

__slots__

现在我们终于明白了，动态语言与静态语言的不同

动态语言：可以在运行的过程中，修改代码

静态语言：编译时已经确定好代码，运行过程中不能修改

如果我们想要限制实例的属性怎么办？比如，只允许对Person实例添加name和age属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的__slots__变量，来限制该class实例能添加的属性：

```
>>> class Person(object):
...     __slots__ = ("name", "age")

>>> P = Person()
>>> P.name = "老王"
>>> P.age = 20
>>> P.score = 100
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
AttributeError: Person instance has no attribute 'score'
>>>
```

注意：

- 使用__slots__要注意，__slots__定义的属性仅对当前类实例起作用，对继承的子类是不起作用的

```
In [67]: class Test(Person):
...     pass
...     
```

```
In [68]: t = Test()
```

```
In [69]: t.score = 100
```

生成器

1. 什么是生成器

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。**

2. 创建生成器方法1

要创建一个生成器，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()

```
In [15]: L = [ x*2 for x in range(5)]  
  
In [16]: L  
Out[16]: [0, 2, 4, 6, 8]  
  
In [17]: G = ( x*2 for x in range(5))  
  
In [18]: G  
Out[18]: <generator object <genexpr> at 0x7f626c132db0>  
  
In [19]:
```

创建 L 和 G 的区别仅在于最外层的 [] 和 ()， L 是一个列表，而 G 是一个生成器。我们可以直接打印出L的每一个元素，但我们怎么打印出G的每一个元素呢？如果要一个一个打印出来，**可以通过 next() 函数获得生成器的下一个返回值：**

```
In [19]: next(G)
```

```
Out[19]: 0
```

```
In [20]: next(G)
```

```
Out[20]: 2
```

```
In [21]: next(G)
```

```
Out[21]: 4
```

```
In [22]: next(G)
```

```
Out[22]: 6
```

```
In [23]: next(G)
```

```
Out[23]: 8
```

```
In [24]: next(G)
```

```
-----  
StopIteration
```

```
Traceback (most recent
```

```
<ipython-input-24-380e167d6934> in <module>()
```

```
----> 1 next(G)
```

```
StopIteration:
```

```
In [25]:
```

```
In [26]: G = ( x**2 for x in range(5))
```

```
In [27]: for x in G:  
....:     print(x)
```

```
....:  
0  
2  
4  
6  
8
```

In [28]:

生成器保存的是算法，每次调用 `next(G)`，就计算出 `G` 的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出 `StopIteration` 的异常。当然，这种不断调用 `next()` 实在是太变态了，正确的方法是使用 `for` 循环，因为生成器也是可迭代对象。所以，我们创建了一个生成器后，基本上永远不会调用 `next()`，而是通过 `for` 循环来迭代它，并且不需要关心 `StopIteration` 异常。

3. 创建生成器方法2

`generator`非常强大。如果推算的算法比较复杂，用类似列表生成式的 `for` 循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
In [28]: def fib(times):  
....:     n = 0  
....:     a, b = 0, 1  
....:     while n<times:  
....:         print(b)  
....:         a, b = b, a+b  
....:         n+=1
```

```
....:     return 'done'  
....:  
  
In [29]: fib(5)  
1  
1  
2  
3  
5  
Out[29]: 'done'
```

仔细观察，可以看出，fib函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把fib函数变成generator，只需要把print(b)改为yield b就可以了：

```
In [30]: def fib(times):  
....:     n = 0  
....:     a, b = 0, 1  
....:     while n < times:  
....:         yield b  
....:         a, b = b, a+b  
....:         n+=1  
....:     return 'done'  
....:
```

```
In [31]: F = fib(5)
```

```
In [32]: next(F)  
Out[32]: 1
```

```
In [33]: next(F)  
Out[33]: 1
```

```
In [34]: next(F)
```

```
Out[34]: 2
```

```
In [35]: next(F)
```

```
Out[35]: 3
```

```
In [36]: next(F)
```

```
Out[36]: 5
```

```
In [37]: next(F)
```

```
-----  
StopIteration
```

```
Traceback (most recent
```

```
<ipython-input-37-8c2b02b4361a> in <module>()
```

```
----> 1 next(F)
```

```
StopIteration: done
```

在上面fib 的例子，我们在循环过程中不断调用 yield ，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。同样的，把函数改成generator后，我们基本上从来不会用 next() 来获取下一个返回值，而是直接使用 for 循环来迭代：

```
In [38]: for n in fib(5):
```

```
....:     print(n)
```

```
....:
```

```
1
```

```
1
```

```
2
```

```
3
```

```
5
```

```
In [39]:
```

但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

```
In [39]: g = fib(5)

In [40]: while True:
....:     try:
....:         x = next(g)
....:         print("value:%d"%x)
....:     except StopIteration as e:
....:         print("生成器返回值:%s"%e.value)
....:         break
....:
value:1
value:1
value:2
value:3
value:5
生成器返回值:done

In [41]:
```

4. send

例子：执行到yield时，gen函数作用暂时保存，返回i的值;temp接收下次c.send("python")，send发送过来的值，c.next()等价c.send(None)

```
In [10]: def gen():
....:     i = 0
....:     while i<5:
....:         temp = yield i
....:         print(temp)
....:         i+=1
....:
```

使用next函数

```
In [11]: f = gen()
```

```
In [12]: next(f)
```

```
Out[12]: 0
```

```
In [13]: next(f)
```

```
None
```

```
Out[13]: 1
```

```
In [14]: next(f)
```

```
None
```

```
Out[14]: 2
```

```
In [15]: next(f)
```

```
None
```

```
Out[15]: 3
```

```
In [16]: next(f)
```

```
None
```

```
Out[16]: 4
```

```
In [17]: next(f)
```

```
None
```

```
-----  
StopIteration
```

```
Traceback (most recent  
<ipython-input-17-468f0afdf1b9> in <module>()  
----> 1 next(f)
```

```
StopIteration:
```

使用 __next__() 方法

```
In [18]: f = gen()
```

```
In [19]: f.__next__()  
Out[19]: 0
```

```
In [20]: f.__next__()  
None  
Out[20]: 1
```

```
In [21]: f.__next__()  
None  
Out[21]: 2
```

```
In [22]: f.__next__()  
None  
Out[22]: 3
```

```
In [23]: f.__next__()  
None  
Out[23]: 4
```

```
In [24]: f.__next__()  
None
```

```
-----  
StopIteration                                Traceback (most recent  
<ipython-input-24-39ec527346a9> in <module>()  
----> 1 f.__next__()
```

```
StopIteration:
```

使用send

```
In [43]: f = gen()
```

```
In [44]: f.__next__()  
Out[44]: 0
```

```
In [45]: f.send('haha')
```

```
haha
```

```
Out[45]: 1
```

```
In [46]: f.__next__()
```

```
None
```

```
Out[46]: 2
```

```
In [47]: f.send('haha')
```

```
haha
```

```
Out[47]: 3
```

```
In [48]:
```

总结

生成器是这样一个函数，它记住上一次返回时在函数体中的位置。对生成器函数的第二次（或第 n 次）调用跳转至该函数中间，而上次调用的所有局部变量都保持不变。

生成器不仅“记住”了它数据状态；生成器还“记住”了它在流控制构造（在命令式编程中，这种构造不只是数据值）中的位置。

生成器的特点：

1. 节约内存
2. 迭代到下一次的调用时，所使用的参数都是第一次所保留下的，也就是说，在整个所有函数调用的参数都是第一次所调用时保留的，而不是新创建的

迭代器

迭代是访问集合元素的一种方式。迭代器是一个可以记住遍历的位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

1. 可迭代对象

以直接作用于 for 循环的数据类型有以下几种：

一类是集合数据类型，如 list 、 tuple 、 dict 、 set 、 str 等；

一类是 generator ，包括生成器和带 yield 的generator function。

这些可以直接作用于 for 循环的对象统称为可迭代对象： Iterable 。

2. 判断是否可以迭代

可以使用 isinstance() 判断一个对象是否是 Iterable 对象：

```
In [50]: from collections import Iterable

In [51]: isinstance([], Iterable)
Out[51]: True

In [52]: isinstance({}, Iterable)
Out[52]: True

In [53]: isinstance('abc', Iterable)
Out[53]: True

In [54]: isinstance((x for x in range(10)), Iterable)
Out[54]: True
```

```
In [55]: isinstance(100, Iterable)
Out[55]: False
```

而生成器不但可以作用于 for 循环，还可以被 next() 函数不断调用并返回下一个值，直到最后抛出 StopIteration 错误表示无法继续返回下一个值了。

3. 迭代器

可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。

可以使用 isinstance() 判断一个对象是否是 Iterator 对象：

```
In [56]: from collections import Iterator

In [57]: isinstance((x for x in range(10)), Iterator)
Out[57]: True

In [58]: isinstance([], Iterator)
Out[58]: False

In [59]: isinstance({}, Iterator)
Out[59]: False

In [60]: isinstance('abc', Iterator)
Out[60]: False

In [61]: isinstance(100, Iterator)
Out[61]: False
```

4. iter()函数

生成器都是 Iterator 对象，但 list 、 dict 、 str 虽然是 Iterable ，却不是 Iterator 。

把 list 、 dict 、 str 等 Iterable 变成 Iterator 可以使用 iter() 函数：

```
In [62]: isinstance(iter([]), Iterator)
Out[62]: True

In [63]: isinstance(iter('abc'), Iterator)
Out[63]: True
```

总结

- 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
- 集合数据类型如 list 、 dict 、 str 等是 Iterable 但不是 Iterator ，不过可以通过 iter() 函数获得一个 Iterator 对象。

闭包

1. 函数引用

```

def test1():
    print("--- in test1 func----")

#调用函数
test1()

#引用函数
ret = test1

print(id(ret))
print(id(test1))

#通过引用调用函数
ret()

```

运行结果：

```

--- in test1 func----
140212571149040
140212571149040
--- in test1 func----

```

2. 什么是闭包

```

#定义一个函数
def test(number):

    #在函数内部再定义一个函数，并且这个函数用到了外边函数的变量，那么将这个
    def test_in(number_in):

```

```
    print("in test_in 函数, number_in is %d"%number_in)
    return number+number_in
#其实这里返回的就是闭包的结果
return test_in

#给test函数赋值, 这个20就是给参数number
ret = test(20)

#注意这里的100其实给参数number_in
print(ret(100))

#注意这里的200其实给参数number_in
print(ret(200))
```

运行结果：

```
in test_in 函数, number_in is 100
120

in test_in 函数, number_in is 200
220
```

3. 闭包再理解

内部函数对外部函数作用域里变量的引用（非全局变量），则称内部函数为闭包。

```
# closure.py

def counter(start=0):
    count=[start]
    def incr():
        count[0] += 1
```

```
        return count[0]
    return incr
```

启动python解释器

```
>>>import closeure
>>>c1=closeure.counter(5)
>>>print(c1())
6
>>>print(c1())
7
>>>c2=closeure.counter(100)
>>>print(c2())
101
>>>print(c2())
102
```

nonlocal访问外部函数的局部变量(python3)

```
def counter(start=0):
    def incr():
        nonlocal start
        start += 1
        return start
    return incr

c1 = counter(5)
print(c1())
print(c1())

c2 = counter(50)
print(c2())
print(c2())

print(c1())
print(c1())
```

```
print(c2())
print(c2())
```

4. 看一个闭包的实际例子：

```
def line_conf(a, b):
    def line(x):
        return a*x + b
    return line

line1 = line_conf(1, 1)
line2 = line_conf(4, 5)
print(line1(5))
print(line2(5))
```

这个例子中，函数line与变量a,b构成闭包。在创建闭包的时候，我们通过line_conf的参数a,b说明了这两个变量的取值，这样，我们就确定了函数的最终形式($y = x + 1$ 和 $y = 4x + 5$)。我们只需要变换参数a,b，就可以获得不同的直线表达函数。由此，我们可以看到，闭包也具有提高代码可复用性的作用。

如果没有闭包，我们需要每次创建直线函数的时候同时说明a,b,x。这样，我们就需要更多的参数传递，也减少了代码的可移植性。

闭包思考：

1. 闭包似优化了变量，原来需要类对象完成的工作，闭包也可以完成
2. 由于闭包引用了外部函数的局部变量，则外部函数的局部变量没有及时释放，消耗内

装饰器

装饰器是程序开发中经常会用到的一个功能，用好了装饰器，开发效率如虎添翼，所以这也是Python面试中必问的问题，但对于好多初次接触这个知识的人来讲，这个功能有点绕，自学时直接绕过去了，然后面试问到了就挂了，因为装饰器是程序开发的基础知识，这个都不会，别跟人家说你会Python，看了下面的文章，保证你学会装饰器。

1、先明白这段代码

```
##### 第一波 #####
def foo():
    print('foo')

foo      #表示是函数
foo()    #表示执行foo函数

##### 第二波 #####
def foo():
    print('foo')

foo = lambda x: x + 1

foo()    # 执行下面的lambda表达式，而不再是原来的foo函数，因为foo这个名字
```

2、需求来了

初创公司有N个业务部门，1个基础平台部门，基础平台负责提供底层的功能，如：数据库操作、redis调用、监控API等功能。业务部门使用基础功能时，只需调用基础平台提供的功能即可。如下：

```
##### 基础平台提供的功能如下 #####
def f1():
    print('f1')

def f2():
    print('f2')

def f3():
    print('f3')

def f4():
    print('f4')

##### 业务部门A 调用基础平台提供的功能 #####
f1()
f2()
f3()
f4()

##### 业务部门B 调用基础平台提供的功能 #####
f1()
f2()
f3()
f4()
```

目前公司有条不紊的进行着，但是，以前基础平台的开发人员在写代码时候没有关注验证相关的问题，即：基础平台的提供的功能可以被任何人使用。现在需要对基础平台的所有功能进行重构，为平台提供的所有功能添加验证机制，即：执行功能前，先进行验证。

老大把工作交给 Low B，他是这么做的：

跟每个业务部门交涉，每个业务部门自己写代码，调用基础平台的功能之前先验证。唉，这样一来基础平台就不需要做任何修改了。太棒了，有充足的时间泡妹子...

当天Low B 被开除了...

老大把工作交给 Low BB，他是这么做的：

```
##### 基础平台提供的功能如下 ######
```

```
def f1():
    # 验证1
    # 验证2
    # 验证3
    print('f1')

def f2():
    # 验证1
    # 验证2
    # 验证3
    print('f2')

def f3():
    # 验证1
    # 验证2
    # 验证3
    print('f3')

def f4():
    # 验证1
    # 验证2
    # 验证3
    print('f4')
```

```
##### 业务部门不变 #####
### 业务部门A 调用基础平台提供的功能###
```

```
f1()  
f2()  
f3()  
f4()  
  
#### 业务部门B 调用基础平台提供的功能 ####
```

```
f1()  
f2()  
f3()  
f4()
```

过了一周 Low BB 被开除了...

老大把工作交给 Low BBB，他是这么做的：

只对基础平台的代码进行重构，其他业务部门无需做任何修改

```
##### 基础平台提供的功能如下 #####
```

```
def check_login():  
    # 验证1  
    # 验证2  
    # 验证3  
    pass
```

```
def f1():  
  
    check_login()  
  
    print('f1')  
  
def f2():  
  
    check_login()
```

```
print('f2')

def f3():
    check_login()

    print('f3')

def f4():
    check_login()

    print('f4')
```

老大看了下Low BBB 的实现，嘴角漏出了一丝的欣慰的笑，语重心长的跟 Low BBB聊了个天：

老大说：

写代码要遵循 **开放封闭 原则**，虽然在这个原则是用的面向对象开发，但是也适用于函数式编程，简单来说，它规定已经实现的功能代码不允许被修改，但可以被扩展，即：

- 封闭：已实现的功能代码块
- 开放：对扩展开发

如果将开放封闭原则应用在上述需求中，那么就不允许在函数 f1 、f2、f3、f4的内部进行修改代码，老板就给了Low BBB一个实现方案：

```
def w1(func):
    def inner():
        # 验证1
        # 验证2
        # 验证3
        func()
    return inner
```

```
@w1
def f1():
    print('f1')
@w1
def f2():
    print('f2')
@w1
def f3():
    print('f3')
@w1
def f4():
    print('f4')
```

对于上述代码，也是仅仅对基础平台的代码进行修改，就可以实现在其他人调用函数 f1 f2 f3 f4 之前都进行【验证】操作，并且其他业务部门无需做任何操作。

Low BBB心惊胆战的问了下，这段代码的内部执行原理是什么呢？

老大正要生气，突然Low BBB的手机掉到地上，恰巧屏保就是Low BBB的女友照片，老大一看一紧一抖，喜笑颜开，决定和Low BBB交个好朋友。

详细的开始讲解了：

单独以f1为例：

```
def w1(func):
    def inner():
        # 验证1
        # 验证2
        # 验证3
        func()
    return inner

@w1
def f1():
    print('f1')
```

python解释器就会从上到下解释代码，步骤如下：

1. def w1(func): ==>将w1函数加载到内存
2. @w1

没错， 从表面上看解释器仅仅会解释这两句代码，因为函数在 没有被调用之前其内部代码不会被执行。

从表面上看解释器着实会执行这两句，但是 @w1 这一句代码里却有大文章， @函数名 是python的一种语法糖。

上例@w1内部会执行一下操作：

执行w1函数

执行w1函数，并将 @w1 下面的函数作为w1函数的参数，即：@w1 等价于 w1(f1) 所以，内部就会去执行：

```
def inner():  
    #验证 1  
    #验证 2  
    #验证 3  
    f1()      # func是参数，此时 func 等于 f1  
return inner# 返回的 inner, inner代表的是函数，非执行函数，其实就
```

w1的返回值

将执行完的w1函数返回值 赋值 给@w1下面的函数的函数名f1 即将w1的返回值再重新赋值给 f1，即：

```
新f1 = def inner():  
    #验证 1  
    #验证 2  
    #验证 3  
    原来f1()
```

```
    return inner
```

所以，以后业务部门想要执行 f1 函数时，就会执行 新f1 函数，在新f1 函数内部先执行验证，再执行原来的f1函数，然后将原来f1 函数的返回值返回给了业务调用者。

如此一来，即执行了验证的功能，又执行了原来f1函数的内容，并将原f1函数返回值 返回给业务调用着

Low BBB 你明白了吗？要是没明白的话，我晚上去你家帮你解决吧！！！

3. 再议装饰器

```
#定义函数：完成包裹数据
def makeBold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped

#定义函数：完成包裹数据
def makeItalic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped

@makeBold
def test1():
    return "hello world-1"

@makeItalic
def test2():
    return "hello world-2"

@makeBold
@makeItalic
```

```
def test3():
    return "hello world-3"

print(test1()))
print(test2()))
print(test3()))
```

运行结果:

```
<b>hello world-1</b>
<i>hello world-2</i>
<b><i>hello world-3</i></b>
```

4. 装饰器(decorator)功能

1. 引入日志
2. 函数执行时间统计
3. 执行函数前预备处理
4. 执行函数后清理功能
5. 权限校验等场景
6. 缓存

5. 装饰器示例

例1:无参数的函数

```
from time import ctime, sleep

def timefun(func):
    def wrappedfunc():
        print("%s called at %s"%(func.__name__, ctime()))
        func()
```

```
return wrappedfunc

@timefun
def foo():
    print("I am foo")

foo()
sleep(2)
foo()
```

上面代码理解装饰器执行行为可理解成

```
foo = timefun(foo)
#foo先作为参数赋值给func后, foo接收指向timefun返回的wrappedfunc
foo()
#调用foo(), 即等价调用wrappedfunc()
#内部函数wrappedfunc被引用, 所以外部函数的func变量(自由变量)并没有释放
#func里保存的是原foo函数对象
```

例2:被装饰的函数有参数

```
from time import ctime, sleep

def timefun(func):
    def wrappedfunc(a, b):
        print("%s called at %s"%(func.__name__, ctime()))
        print(a, b)
        func(a, b)
    return wrappedfunc

@timefun
def foo(a, b):
    print(a+b)

foo(3,5)
sleep(2)
```

```
foo(2, 4)
```

例3:被装饰的函数有不定长参数

```
from time import ctime, sleep

def timefun(func):
    def wrappedfunc(*args, **kwargs):
        print("%s called at %s"%(func.__name__, ctime()))
        func(*args, **kwargs)
    return wrappedfunc

@timefun
def foo(a, b, c):
    print(a+b+c)

foo(3,5,7)
sleep(2)
foo(2,4,9)
```

例4:装饰器中的return

```
from time import ctime, sleep

def timefun(func):
    def wrappedfunc():
        print("%s called at %s"%(func.__name__, ctime()))
        func()
    return wrappedfunc

@timefun
def foo():
    print("I am foo")

@timefun
```

```
def getInfo():
    return '----hahah---'

foo()
sleep(2)
foo()

print(getInfo())
```

执行结果：

```
foo called at Fri Nov  4 21:55:35 2016
I am foo
foo called at Fri Nov  4 21:55:37 2016
I am foo
getInfo called at Fri Nov  4 21:55:37 2016
None
```

如果修改装饰器为 `return func()`，则运行结果：

```
foo called at Fri Nov  4 21:55:57 2016
I am foo
foo called at Fri Nov  4 21:55:59 2016
I am foo
getInfo called at Fri Nov  4 21:55:59 2016
----hahah---
```

总结：

- 一般情况下为了让装饰器更通用，可以有return

例5：装饰器带参数，在原有装饰器的基础上，设置外部变量

```
#decorator2.py

from time import ctime, sleep

def timefun_arg(pre="hello"):
    def timefun(func):
        def wrappedfunc():
            print("%s called at %s %s"%(func.__name__, ctime(),
                return func()
        return wrappedfunc
    return timefun

@timefun_arg("itcast")
def foo():
    print("I am foo")

@timefun_arg("python")
def too():
    print("I am too")

foo()
sleep(2)
foo()

too()
sleep(2)
too()
```

可以理解为

```
foo()==timefun_arg("itcast")(foo)()
```

例6：类装饰器（扩展，非重点）

装饰器函数其实是一个接口约束，它必须接受一个callable对象作为参数，然后返回一个callable对象。在Python中一般callable对象都是函数，但也有例外。只要某个对象重写了 `__call__()` 方法，那么这个对象就是 callable的。

```
class Test():
    def __call__(self):
        print('call me!')

t = Test()
t() # call me
```

类装饰器demo

```
class Test(object):
    def __init__(self, func):
        print("---初始化---")
        print("func name is %s" % func.__name__)
        self.__func = func
    def __call__(self):
        print("---装饰器中的功能---")
        self.__func()

#说明:
```

```
#1. 当用Test来装作装饰器对test函数进行装饰的时候，首先会创建Test的实例对象
#   并且会把test这个函数名当做参数传递到__init__方法中
#   即在__init__方法中的func变量指向了test函数体
#
#2. test函数相当于指向了用Test创建出来的实例对象
#
#3. 当在使用test()进行调用时，就相当于让这个对象()，因此会调用这个对象的__
#
#4. 为了能够在__call__方法中调用原来test指向的函数体，所以在__init__方法
#   所以才有了self.__func = func这句代码，从而在调用__call__方法中能够
@Test
def test():
```

```
print("----test---")
test()
showpy()#如果把这句话注释，重新运行程序，依然会看到"--初始化--"
```

运行结果如下：

```
--初始化---
func name is test
---装饰器中的功能---
----test---
```


import导入模块

1. import 搜索路径

```
import sys  
sys.path
```

[In [35]: import sys

[In [36]: sys.path

Out[36]:

```
['',  
 '/usr/bin',  
 '/usr/lib/python35.zip',  
 '/usr/lib/python3.5',  
 '/usr/lib/python3.5/plat-x86_64-linux-gnu',  
 '/usr/lib/python3.5/lib-dynload',  
 '/usr/local/lib/python3.5/dist-packages',  
 '/usr/lib/python3/dist-packages',  
 '/usr/lib/python3/dist-packages/IPython/extensions',  
 '/home/python/.ipython']
```

路径搜索

- 从上面列出的目录里依次查找要导入的模块文件
- '' 表示当前路径

程序执行时导入模块路径

```
sys.path.append('/home/itcast/xxx')  
sys.path.insert(0, '/home/itcast/xxx')      #可以确保先搜索这个路径
```

In [37]: sys.path.insert(0, "/home/python/xxxx")

In [38]: sys.path

Out[38]:

```
['/home/python/yyyy',
 '',
 '/usr/bin',
 '/usr/lib/python35.zip',
 '/usr/lib/python3.5',
 '/usr/lib/python3.5/plat-x86_64-linux-gnu',
 '/usr/lib/python3.5/lib-dynload',
 '/usr/local/lib/python3.5/dist-packages',
 '/usr/lib/python3/dist-packages',
 '/usr/lib/python3/dist-packages/IPython/extensions',
 '/home/python/.ipython']
```

2. 重新导入模块

模块被导入后，`import module` 不能重新导入模块，重新导入需用

- 测试模块内容

```
1 def test():
2     print("----1---")
3
4
5
6
```

`reload_test.py`

- 调用模块中的方法

```
[In 8]: from imp import *
```

```
[In 9]: import reload_test
```

```
[In 10]: reload_test.test()
----1---
```

- 修改测试模块

```
1 def test():
2     print("----2---")
3
4
5
```

- 重新加载模块

```
[In [8]: from imp import *
[In [9]: import reload_test
[In [10]: reload_test.test()
-----1-----

[In [11]: import reload_test
[In [12]: reload_test.test()
-----1-----

[In [13]: reload(reload_test) 重新加载模块
Out[13]: <module 'reload_test' from 'reload_test.py'>
[In [14]: reload_test.test()
-----2-----
```

循环导入

1. 什么是循环导入

a.py

```
from b import b

print '-----this is module a.py-----'
def a():
    print("hello, a")
    b()

a()
```

b.py

```
from a import a

print '-----this is module b.py-----'
def b():
    print("hello, b")

def c():
    a()
c()
```

运行python a.py

```
python@ubuntu:~/workspace/test$ python a.py
Traceback (most recent call last):
  File "a.py", line 1, in <module>
    from b import b
  File "/home/python/workspace/test/b.py", line 1, in <module>
    from a import a
  File "/home/python/workspace/test/a.py", line 1, in <module>
    from b import b
ImportError: cannot import name b
python@ubuntu:~/workspace/test$ █
```

2. 怎样避免循环导入

1. 程序设计上分层，降低耦合
2. 导入语句放在后面需要导入时再导入，例如放在函数体内导入

作用域

什么是命名空间

比如有一个学校，有10个班级，在7班和8班中都有一个叫“小王”的同学，如果在学校的广播中呼叫“小王”时，7班和8班中的这2个人就纳闷了，你是喊谁呢！！！如果是“7班的小王”的话，那么就很明确了，那么此时的7班就是小王所在的范围，即命名空间

globals、locals

在之前学习变量的作用域时，经常会提到局部变量和全局变量，之所以称之为局部、全局，就是因为他们的自作用的区域不同，这就是作用域

- locals

```
[In [6]: test()

[In [7]: A = 100

[In [8]: B = 200

[In [9]: def test():
[...:     a = 11
[...:     b = 22
[...:     print(locals())
[...:

[In [10]: test()
{'a': 11, 'b': 22}
```

- globals

```

>>> A = 100
>>> B = 200
>>> def test():
...     a = 11
...     b = 22
...     print(locals())
...
>>> globals()
{'test': <function test at 0x7fbbae3109d8>, '__loader__': <class '__frozen_importlib.BuiltinImporter'>, '__name__': '__main__', '__doc__': None, '__spec__': None, '__builtins__': <module 'builtins' (built-in)>, 'B': 200, '__package__': None, 'A': 100}
>>>

```

LEGB 规则

Python 使用 LEGB 的顺序来查找一个符号对应的对象

```
locals -> enclosing function -> globals -> builtins
```

- `locals`, 当前所在命名空间（如函数、模块），函数的参数也属于命名空间内的变量
- `enclosing`, 外部嵌套函数的命名空间（闭包中常见）

```

def fun1():
    a = 10
    def fun2():
        # a 位于外部嵌套函数的命名空间
        print(a)

```

- `globals`, 全局变量，函数定义所在模块的命名空间

```

a = 1
def fun():
    # 需要通过 global 指令来声明全局变量
    global a
    # 修改全局变量，而不是创建一个新的 local 变量
    a = 2

```

- `builtins`, 内建模块的命名空间。

Python 在启动的时候会自动为我们载入很多内建的函数、类，
比如 `dict`, `list`, `type`, `print`, 这些都位于 `__builtin__` 模块中，

可以使用 `dir(__builtin__)` 来查看。

这也是为什么我们在没有 `import`任何模块的情况下，

就能使用这么多丰富的函数和功能了。

在Python中，有一个内建模块，该模块中有一些常用函数；在Python启动后，且没有执行程序员所写的任何代码前，Python会首先加载该内建函数到内存。

另外，该内建模块中的功能可以直接使用，不用在其前添加内建模块前缀，

其原因是对函数、变量、类等标识符的查找是按LEGB法则，其中B即代表内建模块。比如：内建模块中有一个`abs()`函数，其功能求绝对值，如`abs(-20)`将返回20。

==、 is

```
[In [70]: a = [11,22,33]
[In [71]: b = a
[In [72]: a == b
Out[72]: True
[In [73]: a is b
Out[73]: True
[In [74]: c = copy.deepcopy(a)
[In [75]: a == c
Out[75]: True
[In [76]: a is c
Out[76]: False
```

总结

- is 是比较两个引用是否指向了同一个对象（引用比较）。
- == 是比较两个对象是否相等。

深拷贝、浅拷贝

1. 浅拷贝

- 浅拷贝是对于一个对象的顶层拷贝

通俗的理解是：拷贝了引用，并没有拷贝内容

[In [26]:

[In [26]: a = [11,22,33]

[In [27]: id(a)
Out[27]: 139706275415752

[In [28]: b = a

[In [29]: id(b)
Out[29]: 139706275415752

[In [30]: a.append(44)

[In [31]: a
Out[31]: [11, 22, 33, 44]

[In [32]: b
Out[32]: [11, 22, 33, 44]

In [33]:

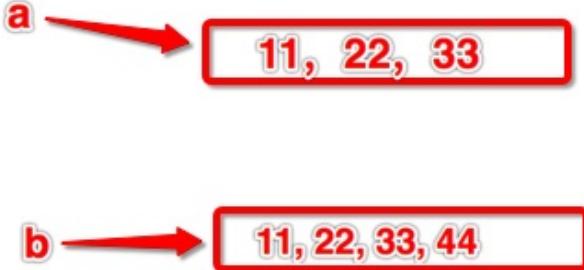


```
[In [33]: a = {"name":"xiaowang"}  
[In [34]: id(a)  
Out[34]: 139706275437128  
  
[In [35]: b = a  
  
[In [36]: id(b)  
Out[36]: 139706275437128  
  
[In [37]: a['id'] = 100  
  
[In [38]: a  
Out[38]: {'name': 'xiaowang', 'id': 100}  
  
[In [39]: b  
Out[39]: {'name': 'xiaowang', 'id': 100}  
  
In [40]: ]
```

2. 深拷贝

- 深拷贝是对于一个对象所有层次的拷贝(递归)

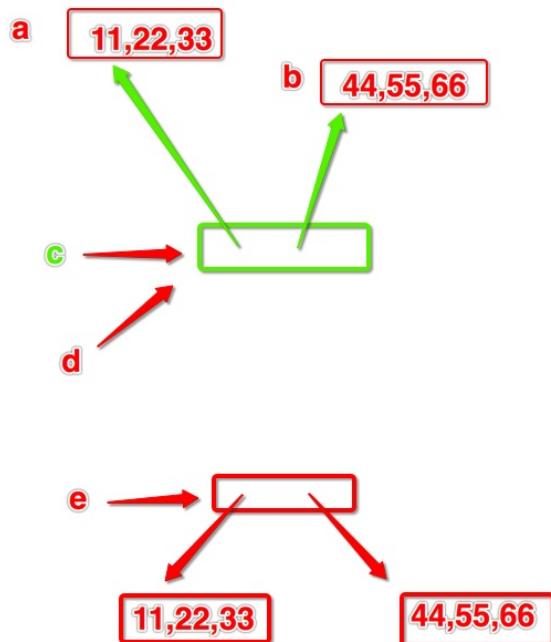
```
[In [49]: import copy  
  
[In [50]: a = [11,22,33]  
  
[In [51]: id(a)  
Out[51]: 139706274929224  
  
[In [52]: b = copy.deepcopy(b)  
  
[In [53]: id(b)  
Out[53]: 139706274930312  
  
[In [54]: a.append(44)  
  
[In [55]: a  
Out[55]: [11, 22, 33, 44]  
  
[In [56]: b  
Out[56]: [11, 22, 33]
```



进一步理解拷贝

```
In [57]: a = [11,22,33]
In [58]: b = [44,55,66]
In [59]: c = [a, b]
In [60]: d = c
In [61]: id(c)
Out[61]: 139706275605512
In [62]: id(d)
Out[62]: 139706275605512
In [63]: e = copy.deepcopy(c)
In [64]: id(e)
Out[64]: 139706275605704
In [65]: a.append(44)

In [66]: a
Out[66]: [11, 22, 33, 44]
In [67]: b
Out[67]: [44, 55, 66]
In [68]: c
Out[68]: [[11, 22, 33, 44], [44, 55, 66]]
In [69]: e
Out[69]: [[11, 22, 33], [44, 55, 66]]
```



```
In [23]: a = [11,22,33]
In [24]: b = [44,55,66]
In [25]: c = (a,b)
In [26]: e = copy.deepcopy(c)
In [27]: a.append(77)
In [28]: a
Out[28]: [11, 22, 33, 77]
In [29]: b
Out[29]: [44, 55, 66]
In [30]: c
Out[30]: ([11, 22, 33, 77], [44, 55, 66])
In [31]: e
Out[31]: ([11, 22, 33], [44, 55, 66])
```

```
In [32]:  
  
In [32]:  
  
In [32]: f = copy.copy(c)  
  
In [33]: a.append(88)  
  
In [34]: a  
Out[34]: [11, 22, 33, 77, 88]  
  
In [35]: b  
Out[35]: [44, 55, 66]  
  
In [36]: c  
Out[36]: ([11, 22, 33, 77, 88], [44, 55, 66])  
  
In [37]: e  
Out[37]: ([11, 22, 33], [44, 55, 66])  
  
In [38]: f  
Out[38]: ([11, 22, 33, 77, 88], [44, 55, 66])
```

3. 拷贝的其他方式

浅拷贝对不可变类型和可变类型的copy不同

```
In [88]: a = [11, 22, 33]  
  
In [89]: b = copy.copy(a)  
  
In [90]: id(a)  
Out[90]: 59275144  
  
In [91]: id(b)
```

```
Out[91]: 59525600
```

```
In [92]: a.append(44)
```

```
In [93]: a
```

```
Out[93]: [11, 22, 33, 44]
```

```
In [94]: b
```

```
Out[94]: [11, 22, 33]
```

```
In [95]:
```

```
In [95]:
```

```
In [95]: a = (11, 22, 33)
```

```
In [96]: b = copy.copy(a)
```

```
In [97]: id(a)
```

```
Out[97]: 58890680
```

```
In [98]: id(b)
```

```
Out[98]: 58890680
```

- 分片表达式可以赋值一个序列

```
a = "abc"
```

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

- 字典的copy方法可以拷贝一个字典

```
d = dict(name="zhangsan", age=27)
```

```
co = d.copy()
```

- 有些内置函数可以生成拷贝(list)

```
a = list(range(10))  
  
b = list(a)
```

- copy模块中的copy函数

```
import copy  
  
a = (1, 2, 3)  
  
b = copy.copy(a)
```

进制、位运算

1、什么是进制

1) 理解个X进制的概念：

每一位 只允许出现 0~X-1 这几个数字,逢X进一,基是X, 每一位有一个权值大小是X的幂次。 其表示的数值可以写成按位权展开的多项式之和。

十进制: 每一位只允许出现0~9这十个数字,逢十进1,基是十,每一位数字有一个权值大小是十的幂次。 其表示的数值可以写成按位权展开的多项式之和。

十进制	第四位	第三位	第二位	第一位			
				0 零	=0*10^0		
				1 一	=1*10^0		
				9 九	=9*10^0		
			1	0 十	=0*10^0+1*10^1		
			1	9 十九	=9*10^0+1*10^1		
			2	0 二十			
			9	9 九十九			
	1	0	0 一百	=0*10^0+0*10^1+1*10^2			

二进制: 每一位只允许出现0~1这二个数字,逢二进1,基是 二, 每一位数字有一个权值大小是二的幂次。 其表示的数值可以写成按位权展开的多项式之和。

二进制	第四位	第三位	第二位	第一位			
				0 零	=0*2^0		
				1 一	=1*2^0		
			1	0 二	=0*2^0+1*2^1		
			1	1 三	=1*2^0+1*2^1		
	1	0	0 四	=0*2^0+0*2^1+1*2^2			
	1	0	1 五	=1*2^0+0*2^1+1*2^2			
	1	1	0 六	=0*2^0+1*2^1+1*2^2			

八进制:

八进制			第二位	第一位		
				0 零		
				1 一		
				7 七 $=7*8^0$		
			1	0 八 $=0*8^0+1*8^1$		
			1	1 九 $=1*8^0+1*8^1$		
			1	2 十		
			1	3 十一		
			1	7 十五		
			2	0 十六 $=0*8^0+2*8^1$		
			2	1 十七		
			1	0 六十四 $=0*8^0+0*8^1+0*8^2$		

十六进制

十六进制	第五位	第四位	第三位	第二位	第一位	
0-9+abcdef					0 零	
					3 三	
					9 九 $=9*16^0$	
					a 十	
					b 十一	
					f 十五 $=十五*16^0$	
				1	0 十六 $=0*16^0+1*16^1$	
				1	9 二十五	
				1	a 二十六	
				1	f 三十一	
				2	0 三十二 $=0*16^0+2*16^1$	
				3	0 四十八	
				1	0 等于 $=?$	

2)

假如用两个字节表示一个整数，如下：

十进制数字1 的二进制表现形式： 0000 0000 0000 0001

十进制数字2 的二进制表现形式： 0000 0000 0000 0010

如何表示二进制数的正负？

3) 有符号数和无符号数的概念

规则：把二进制数中的最高位（最左边的那位）用作符号位

对于有符号数，最高位被计算机系统规定为符号位(0为正, 1为负)

对于无符号数，最高位被计算机系统规定为数据位

按照这种说法，比如有符号数 +2 -2 的原码形式：

+2 = 0000 0000 0000 0010
-2 = 1000 0000 0000 0010
真值 机器数

+1 = 0000 0000 0000 0001
-1 = 1000 0000 0000 0001

1000 0000 0000 0010

-1+1 的结果？

$-1+1 = 1000\ 0000\ 0000\ 0010 \rightarrow -2$

不等于0，按理说-1+1等于0才对，为什么会是-2呢？

规则

数字在计算机中，是用二进制补码的形式来保存的，因此 $-1 + 1$ 需要按照补码进行相加才是正确的结果

2、原码、反码、补码

1) 如何计算补码？

规则：

正数：原码 = 反码 = 补码

负数：
反码 = 符号位不变，其他位取反
补码 = 反码+1

1 的原码：0000 0000 0000 0001
-1的原码：1000 0000 0000 0001
-1的反码：1111 1111 1111 1110
-1的补码：1111 1111 1111 1111

重新计算 -1+1 结果

```
1111 1111 1111 1111  
0000 0000 0000 0001  
-----  
0000 0000 0000 0000
```



2) 从补码转回原码

负数补码转换原码的规则:

原码 = 补码的符号位不变 --> 数据位取反--> 尾+1

-1的补码:1111 1111 1111 1111

 取反:1000 0000 0000 0000

-1的原码:1000 0000 0000 0001

【了解】

可以把减法用加法来算，只需设计加法器就好了。运算的时候都是用补码去运算的。 $2-1 = 2+(-1)=0000\ 0000\ 0000\ 0010 + 1111\ 1111\ 1111\ 1111$

【了解】

为何要使用原码, 反码和补码 既然原码才是被人脑直接识别并用于计算表示方式, 为何还会有反码和补码呢? 首先, 因为人脑可以知道第一位是符号位, 在计算的时候我们会根据符号位, 选择对应加减, 但是对于计算机, 加减乘数已经是最基础的运算, 要设计的尽量简单。计算机辨别"符号位"显然会让计算机的基础电路设计变得十分复杂! 于是人们想出了将符号位也参与运算的方法. 我们知道, 根据运算法则减去一个正数等于加上一个负数, 即: $1-1 = 1 + (-1) = 0$, 所以机器可以只有加法而没有减法, 这样计算机运算的设计就更简单了. 于是人们开始探索 将符号位参与运算, 并且只保留加法的方法

3. 进制间转换

```
#10进制转为2进制
```

```
>>> bin(10)
'0b1010'
```

```
#2进制转为10进制
```

```
>>> int("1001", 2)
9
```

```
#10进制转为16进制
```

```
>>> hex(10)
'0xa'
```

```
#16进制到10进制
```

```
>>> int('ff', 16)
255
```

```
>>> int('0xab', 16)
```

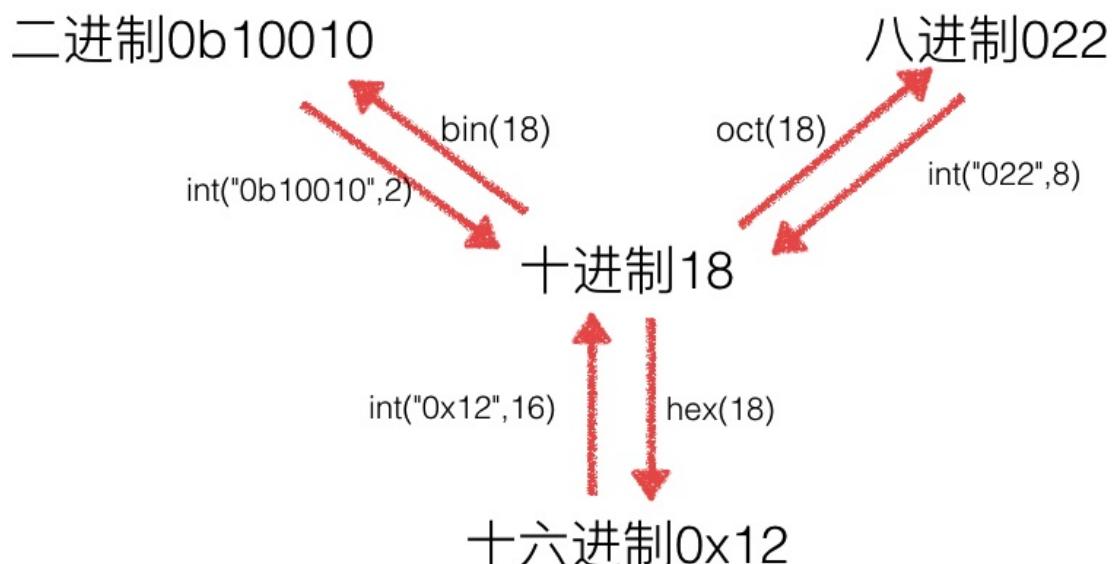
```
171
```

```
#16进制到2进制
```

```
>>> bin(0xa)
'0b1010'
```

```
>>>  
  
#10进制到8进制  
>>> oct(8)  
'010'
```

```
#2进制到16进制  
>>> hex(0b1001)  
'0x9'
```



4. 位运算

看如下示例：

如果有一个十进制数 5，其二进制为： 0000 0101

把所有的数向左移动一位 其结果为： 0000 1010

想一想：二进制 0000 1010 十进制是多少呢？？？ 其答案为10，有没有发现是5的2倍呢！

再假设有一个十进制数 3， 其二进制 为： 0000 0011

把所有的数向左移动一位 其结果为： 0000 0110

二进制0000 0110 的十进制为6，正好也是3的2倍

通过以上2个例子，能够看出，把一个数的各位整体向左移动一个位，就变成原来的2倍

那么在Python中，怎样实现向左移动呢？还有其他的吗？？？

<1>位运算的介绍

- & 按位与
- | 按位或
- ^ 按位异或
- ~ 按位取反
- << 按位左移
- >> 按位右移

用途：直接操作二进制，省内存，效率高

<2>位运算

1) << 按位左移

各二进位全部左移n位，高位丢弃，低位补0

	1000	1000	0101	0000
100	0100	0010	1000	0 <u>000</u>
<u>砍掉</u>			<u>补0</u>	

<< 左移三位

$x \ll n$ 左移 x 的所有二进制位向左移动 n 位, 移出位删掉, 移进的位补零

【注意事项】

- a. 左移1位相当于 乘以2
 - 用途:快速计算一个数乘以2的n次方 ($8 \ll 3$ 等同于 $8 * 2^3$)
- b. 左移可能会改变一个数的正负性

The screenshot shows a Sublime Text editor window titled 'test.py' and a terminal window. The Python code in the editor is:

```
test.py - Sublime Text
test.py
1 #coding=utf-8
2
3
4 num = 0b00000001 0b表示二进制
5
6 print(num)
7
8 num = num << 1
9
10 print(num)
11
12 num = num << 1
13
14 print(num)
15
```

The terminal window shows the execution of the script:

```
python@ubuntu: ~/Desktop
python@ubuntu:~/Desktop$ python3 test.py
1
2
4
python@ubuntu:~/Desktop$
```

2) >> 右移

各二进位全部右移 n 位, 保持符号位不变

$x \gg n$ x 的所有二进制位向右移动 n 位, 移出的位删掉, 移进的位补符号位 右移不会改

【注意事项】

- 右移1位相当于 除以2
- x 右移 n 位就相当于除以2的 n 次方 用途:快速计算一个数除以2的 n 次方 ($8 \gg 3$ 等同于 $8 / 2^3$)

The screenshot shows a Sublime Text window titled 'test.py' containing the following Python code:

```
1 #coding=utf-8
2
3
4 num = 0b00001000
5
6 print(num)
7
8 num = num >> 1
9
10 print(num)
11
12 num = num >> 1
13
14 print(num)
15
```

To the right, a terminal window shows the output of running the script:

```
python@ubuntu:~/Desktop$ python3 test.py
8
4
2
python@ubuntu:~/Desktop$
```

3) & 按位与

全1才1否则0 :只有对应的两个二进位均为1时,结果位才为1,否则为0
用6和3这个例子。不要用9 和13的例子

The screenshot shows a Sublime Text window titled 'test.py' containing the following Python code:

```
1 #coding=utf-8
2
3
4 num = 0b00001000
5
6 print(num)
7
8 num = num & 0b00000001
9
10 print(num)
11
```

To the right, a terminal window shows the output of running the script:

```
python@ubuntu:~/Desktop$ python3 test.py
8
0
python@ubuntu:~/Desktop$
```

4) | 按位或

有1就1 只要对应的二个二进位有一个为1时,结果位就为1,否则为0

The screenshot shows a Sublime Text window titled 'test.py' containing the following Python code:

```
1 #coding=utf-8
2
3
4 num = 0b00001000
5
6 print(num)
7
8 num = num | 0b00000001
9
10 print(num)
11
```

To the right, a terminal window shows the output of running the script:

```
python@ubuntu:~/Desktop$ python3 test.py
8
9
python@ubuntu:~/Desktop$
```

5) ^ 按位异或

不同为1 当对应的二进位相异(不相同)时, 结果为1, 否则为0

The screenshot shows a Sublime Text editor with a file named 'test.py' containing the following code:

```
# coding=utf-8
num = 0b00001000
print(num)
num = num^0b00000100
print(num)
```

Next to it is a terminal window titled 'python@ubuntu: ~/Desktop' showing the output of running the script:

```
python@ubuntu:~/Desktop$ python3 test.py
8
12
python@ubuntu:~/Desktop$
```

6) ~ 取反

$\sim 9 = -10$

The screenshot shows a Sublime Text editor with a file named 'test.py' containing the following code:

```
# coding=utf-8
num = 0b00001000
print(num)
num = ~num
print(num)
```

Next to it is a terminal window titled 'python@ubuntu: ~/Desktop' showing the output of running the script:

```
python@ubuntu:~/Desktop$ python3 test.py
8
-9
python@ubuntu:~/Desktop$
```

【为什么9取反变成了-10的说明】：

9的原码 ==> 0000 1001 因为正数的原码=反码=补码，所以在真正存储的时候就是0000 1001

接下来进行对9的补码进行取反操作

进行取反==> 1111 0110 这就是对9进行了取反之后的补码

既然已经知道了补码，那么接下来只要转换为咱们人能识别的码型就可以，因此按照规则，把这个1111 0110 这个补码 转换为原码即可

符号位不变，其它位取反==> 1000 1001

然后+1， 得到原码 =====>1000 1010 这就是 -10

【扩展】

1)任何数和1进行&操作,得到这个数的最低位 $\text{数字} \& 1 = \text{数字的二进制形式的最低位}$

2)位运算优先级

私有化

- xx: 公有变量
- _x: 单前置下划线,私有化属性或方法, from somemodule import *禁止导入,类对象和子类可以访问
- __xx: 双前置下划线,避免与子类中的属性命名冲突, 无法在外部直接访问(名字重整所以访问不到)
- __xx__:双前后下划线,用户名字空间的魔法对象或属性。例如: __init__, __不要自己发明这样的名字
- xx_:单后置下划线,用于避免与Python关键词的冲突

通过name mangling (名字重整(目的就是以防子类意外重写基类的方法或者属性)如: __Class__object) 机制就可以访问private了。

```
#coding=utf-8

class Person(object):
    def __init__(self, name, age, taste):
        self.name = name
        self._age = age
        self.__taste = taste

    def showperson(self):
        print(self.name)
        print(self._age)
        print(self.__taste)

    def dowork(self):
        self._work()
        self.__away()

    def __work(self):
        print('my __work')
```

```
def __away(self):
    print('my __away')

class Student(Person):
    def construction(self, name, age, taste):
        self.name = name
        self._age = age
        self.__taste = taste

    def showstudent(self):
        print(self.name)
        print(self._age)
        print(self.__taste)

    @staticmethod
    def testbug():
        _Bug.showbug()

#模块内可以访问, 当from cur_module import *时, 不导入
class _Bug(object):
    @staticmethod
    def showbug():
        print("showbug")

s1 = Student('jack', 25, 'football')
s1.showperson()
print('*'*20)

#无法访问__taste, 导致报错
#s1.showstudent()
s1.construction('rose', 30, 'basketball')
s1.showperson()
print('*'*20)

s1.showstudent()
print('*'*20)

Student.testbug()
```

```
python@ubuntu:~/workspace/test$ python t10.py
jack
25
football
*****
rose
30
football
*****
rose
30
basketball
*****
showbug
python@ubuntu:~/workspace/test$ █
```

总结

- 父类中属性名为 `_名字` 的，子类不继承，子类不能访问
- 如果在子类中向 `_名字` 赋值，那么会在子类中定义的一个与父类相同名字的属性
- `_名` 的变量、函数、类在使用 `from xxx import *` 时都不会被导入

属性property

1. 私有属性添加getter和setter方法

```
class Money(object):
    def __init__(self):
        self.__money = 0

    def getMoney(self):
        return self.__money

    def setMoney(self, value):
        if isinstance(value, int):
            self.__money = value
        else:
            print("error:不是整型数字")
```

2. 使用property升级getter和setter方法

```
class Money(object):
    def __init__(self):
        self.__money = 0

    def getMoney(self):
        return self.__money

    def setMoney(self, value):
        if isinstance(value, int):
            self.__money = value
        else:
            print("error:不是整型数字")
money = property(getMoney, setMoney)
```

运行结果：

```
In [1]: from get_set import Money  
  
In [2]:  
  
In [2]: a = Money()  
  
In [3]:  
  
In [3]: a.money  
Out[3]: 0  
  
In [4]: a.money = 100  
  
In [5]: a.money  
Out[5]: 100  
  
In [6]: a.getMoney()  
Out[6]: 100
```

3. 使用property取代getter和setter方法

@property 成为属性函数，可以对属性赋值时做必要的检查，并保证代码的清晰短小，主要有2个作用

- 将方法转换为只读
- 重新实现一个属性的设置和读取方法,可做边界判定

```
class Money(object):  
    def __init__(self):  
        self.__money = 0  
  
    @property  
    def money(self):  
        return self.__money
```

```
@money.setter  
def money(self, value):  
    if isinstance(value, int):  
        self.__money = value  
    else:  
        print("error:不是整型数字")
```

运行结果

```
In [3]: a = Money()
```

```
In [4]:
```

```
In [4]:
```

```
In [4]: a.money
```

```
Out[4]: 0
```

```
In [5]: a.money = 100
```

```
In [6]: a.money
```

```
Out[6]: 100
```

其他的知识点

垃圾回收

1. 小整数对象池

整数在程序中的使用非常广泛，Python为了优化速度，使用了小整数对象池，避免为整数频繁申请和销毁内存空间。

Python 对小整数的定义是 [-5, 257) 这些整数对象是提前建立好的，不会被垃圾回收。在一个 Python 的程序中，所有位于这个范围内的整数使用的都是同一个对象。

同理，单个字母也是这样的。

但是当定义2个相同的字符串时，引用计数为0，触发垃圾回收

2. 大整数对象池

每一个大整数，均创建一个新的对象。

```
In [15]: b=1356
In [16]: id(b)
Out[16]: 39430264

In [17]: a=1356

In [18]: id(a)
Out[18]: 38895736

In [19]: type(a)
Out[19]: int

In [20]: b=a

In [21]: id(b)
Out[21]: 38895736
```

3. intern机制

```
a1 = "HelloWorld"
a2 = "HelloWorld"
a3 = "HelloWorld"
a4 = "HelloWorld"
a5 = "HelloWorld"
a6 = "HelloWorld"
a7 = "HelloWorld"
a8 = "HelloWorld"
a9 = "HelloWorld"
```

python会不会创建9个对象呢？在内存中会不会开辟9个"HelloWorld"的内存空间呢？想一下，如果是这样的话，我们写10000个对象，比如 a1="HelloWorld".....a1000="HelloWorld"，那他岂不是开辟了1000个"HelloWorld"所占的内存空间了呢？如果真这样，内存不就爆了吗？所以 python中有这样一个机制——intern机制，让他只占用一个"HelloWorld"所占的内存空间。靠引用计数去维护何时释放。

```
In [22]: a='abcde'
In [23]: b='abcde'
In [24]: id(b)
Out[24]: 139929937277168

In [25]: id(a)
Out[25]: 139929937277168

In [26]: del(a)

In [27]: del(b)

In [28]: e='abcde'

In [29]: id(e)
Out[29]: 139929937277360
```

总结

- 小整数[-5,257)共用对象，常驻内存
- 单个字符共用对象，常驻内存
- 单个单词，不可修改，默认开启intern机制，共用对象，引用计数为0，则销毁

```
In [53]: a='hello'  
In [54]: b='hello'  
In [55]: id(a)  
Out[55]: 140086726350768  
In [56]: id(b)  
Out[56]: 140086726350768  
In [57]: del(a)  
In [58]: del(b)  
In [59]: c='hello'  
In [60]: id(c)  
Out[60]: 140086726350336
```

- 字符串（含有空格），不可修改，没开启intern机制，不共用对象，引用计数为0，销毁

```
[In [1]: a = "hello world"  
[In [2]: b = "hello world"  
[In [3]: id(a)  
Out[3]: 140678072380016  
[In [4]: id(b)  
Out[4]: 140678072380080  
[In [5]: c = "helloworld"  
[In [6]: d = "helloworld"  
[In [7]: id(c)  
Out[7]: 140678072380400  
[In [8]: id(d)  
Out[8]: 140678072380400
```

是否有空格?

- 大整数不共用内存，引用计数为0，销毁

```
In [15]: b=1356
In [16]: id(b)
Out[16]: 39430264
In [17]: a=1356
In [18]: id(a)
Out[18]: 38895736
In [19]: type(a)
Out[19]: int
In [20]: b=a
In [21]: id(b)
Out[21]: 38895736
```

- 数值类型和字符串类型在 Python 中都是不可变的，这意味着你无法修改这个对象的值，每次对变量的修改，实际上是创建一个新的对象

```
In [62]: id(a)
Out[62]: 37122392
In [63]: a+=1
In [64]: id(a)
Out[64]: 37122368
In [65]: b='hello'
In [66]: id(b)
Out[66]: 140086726350336
In [67]: b='itcat'
In [68]: id(b)
Out[68]: 140086726351392
```

垃圾回收(二)

1. Garbage collection(GC垃圾回收)

现在的高级语言如java, c#等，都采用了垃圾收集机制，而不再是c, c++里用户自己管理维护内存的方式。自己管理内存极其自由，可以任意申请内存，但如同一把双刃剑，为大量内存泄露，悬空指针等bug埋下隐患。对于一个字符串、列表、类甚至数值都是对象，且定位简单易用的语言，自然不会让用户去处理如何分配回收内存的问题。python里也同java一样采用了垃圾收集机制，不过不一样的是：python采用的是引用计数机制为主，标记-清除和分代收集两种机制为辅的策略

引用计数机制：

python里每一个东西都是对象，它们的核心就是一个结构体： PyObject

```
typedef struct_object {
    int ob_refcnt;
    struct_typeobject *ob_type;
} PyObject;
```

PyObject是每个对象必有的内容，其中ob_refcnt就是做为引用计数。当一个对象有新的引用时，它的ob_refcnt就会增加，当引用它的对象被删除，它的ob_refcnt就会减少

```
#define Py_INCREF(op) ((op)->ob_refcnt++) //增加计数
#define Py_DECREF(op) \
    if (--(op)->ob_refcnt != 0) \
        ; \
    else \
        __Py_Dealloc((PyObject *) (op))
```

当引用计数为0时，该对象生命就结束了。

引用计数机制的优点：

- 简单
- 实时性：一旦没有引用，内存就直接释放了。不用像其他机制等到特定时机。实时性还带来一个好处：处理回收内存的时间分摊到了平时。

引用计数机制的缺点：

- 维护引用计数消耗资源
- 循环引用

```
list1 = []
list2 = []
list1.append(list2)
list2.append(list1)
```

list1与list2相互引用，如果不存在其他对象对它们的引用，list1与list2的引用计数也仍然为1，所占用的内存永远无法被回收，这将是致命的。对于如今的强大硬件，缺点1尚可接受，但是循环引用导致内存泄露，注定python还将引入新的回收机制。(标记清除和分代收集)

2. 画说 Ruby 与 Python 垃圾回收

英文原文: [visualizing garbage collection in ruby and python](#)

2.1 应用程序那颗跃动的心

GC系统所承担的工作远比"垃圾回收"多得多。实际上，它们负责三个重要任务。它们

- 为新生成的对象分配内存
- 识别那些垃圾对象，并且
- 从垃圾对象那回收内存。

如果将应用程序比作人的身体：所有你所写的那些优雅的代码，业务逻辑，算法，应该就是大脑。以此类推，垃圾回收机制应该是那个身体器官呢？

(我从RuPy听众那听到了不少有趣的答案：腰子、白血球 :))

我认为垃圾回收就是应用程序那颗跃动的心。像心脏为身体其他器官提供血液和营养物那样，垃圾回收器为你的应用程序提供内存和对象。如果心脏停跳，过不了几秒钟人就完了。如果垃圾回收器停止工作或运行迟缓，像动脉阻塞，你的应用程序效率也会下降，直至最终死掉。

2.2 一个简单的例子

运用实例一贯有助于理论的理解。下面是一个简单类，分别用Python和Ruby写成，我们今天就以此为例：

```
class Node:
    def __init__(self, val):
        self.value = val

print(Node(1))
print(Node(2))
```

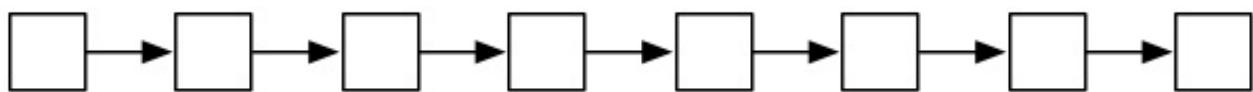
```
class Node
  def initialize(val)
    @value = val
  end
end

p Node.new(1)
p Node.new(2)
```

顺便提一句，两种语言的代码竟能如此相像：Ruby 和 Python 在表达同一事物上真的只是略有不同。但是在这两种语言的内部实现上是否也如此相似呢？

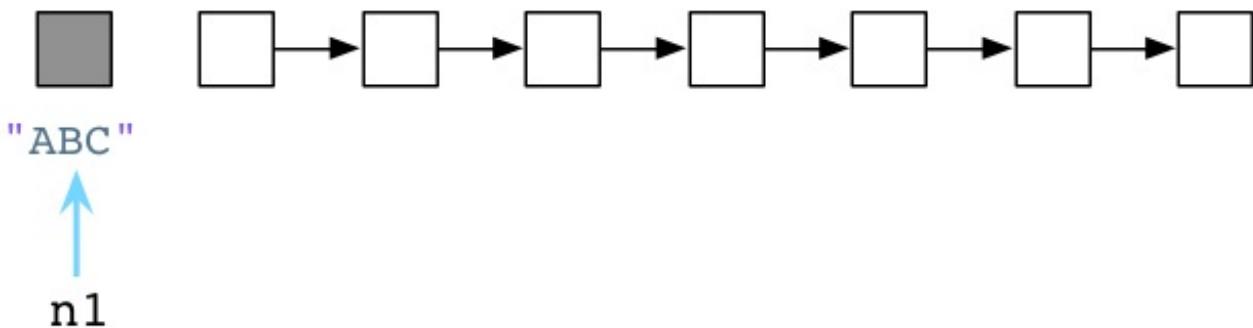
2.3 Ruby 的对象分配

当我们执行上面的Node.new(1)时，Ruby到底做了什么？Ruby是如何为我们创建新的对象的呢？出乎意料的是它做的非常少。实际上，早在代码开始执行前，Ruby就提前创建了成百上千个对象，并把它们串在链表上，名曰：可用列表。下图所示为可用列表的概念图：



想象一下每个白色方格上都标着一个"未使用预创建对象"。当我们调用 Node.new ,Ruby只需取一个预创建对象给我们使用即可：

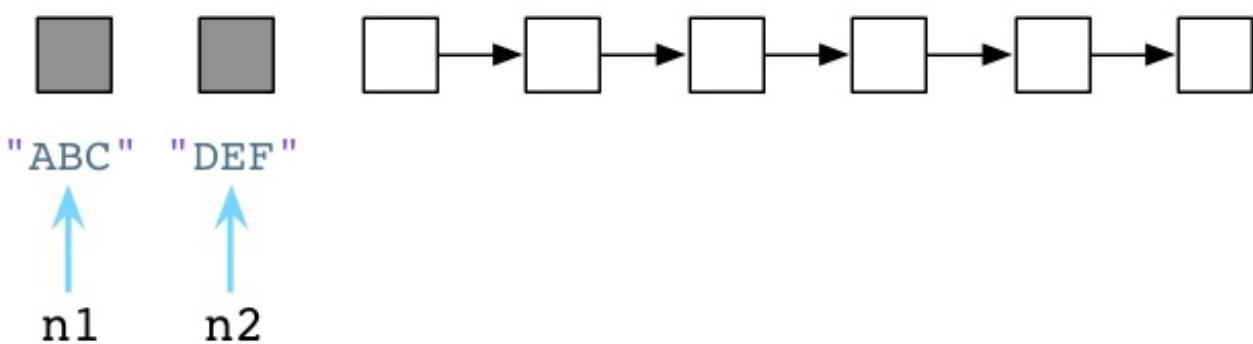
```
n1 = Node.new("ABC")
```



上图中左侧灰格表示我们代码中使用的当前对象，同时其他白格是未使用对象。(请注意：无疑我的示意图是对实际的简化。实际上，Ruby会用另一个对象来装载字符串"ABC",另一个对象装载Node类定义，还有一个对象装载了代码中分析出的抽象语法树，等等)

如果我们再次调用 Node.new, Ruby将递给我们另一个对象：

```
n2 = Node.new("DEF")
```



这个简单的用链表来预分配对象的算法已经发明了超过50年，而发明人这是赫赫有名的计算机科学家John McCarthy，一开始是用Lisp实现的。Lisp不仅是最早的函数式编程语言，在计算机科学领域也有许多创举。其一就是利用垃圾回收机制自动化进行程序内存管理的概念。

标准版的Ruby，也就是众所周知的"Matz's Ruby Interpreter"(MRI),所使用的GC算法与McCarthy在1960年的实现方式很类似。无论好坏，Ruby的垃圾回收机制已经53岁高龄了。像Lisp一样，Ruby预先创建一些对象，然后在你分配新对象或者变量的时候供你使用。

2.4 Python 的对象分配

我们已经了解了Ruby预先创建对象并将它们存放在可用列表中。那Python又怎么样呢？

尽管由于许多原因Python也使用可用列表(用来回收一些特定对象比如 list)，但在为新对象和变量分配内存的方面Python和Ruby是不同的。

例如我们用Python来创建一个Node对象：

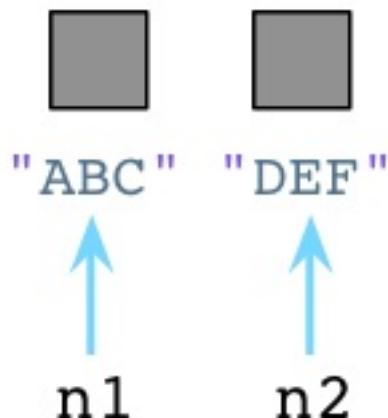
```
n1 = Node("ABC")
```



与Ruby不同，当创建对象时Python立即向操作系统请求内存。(Python实际上实现了一套自己的内存分配系统，在操作系统堆之上提供了一个抽象层。但是我今天不展开说了。)

当我们创建第二个对象的时候，再次像OS请求内存：

```
n2 = Node("DEF")
```



看起来够简单吧，在我们创建对象的时候，Python会花些时间为找到并分配内存。

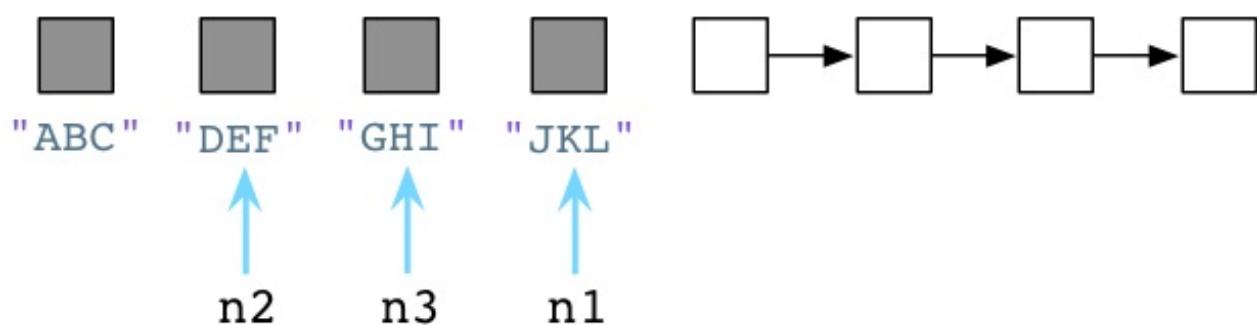
2.5 Ruby 开发者住在凌乱的房间里



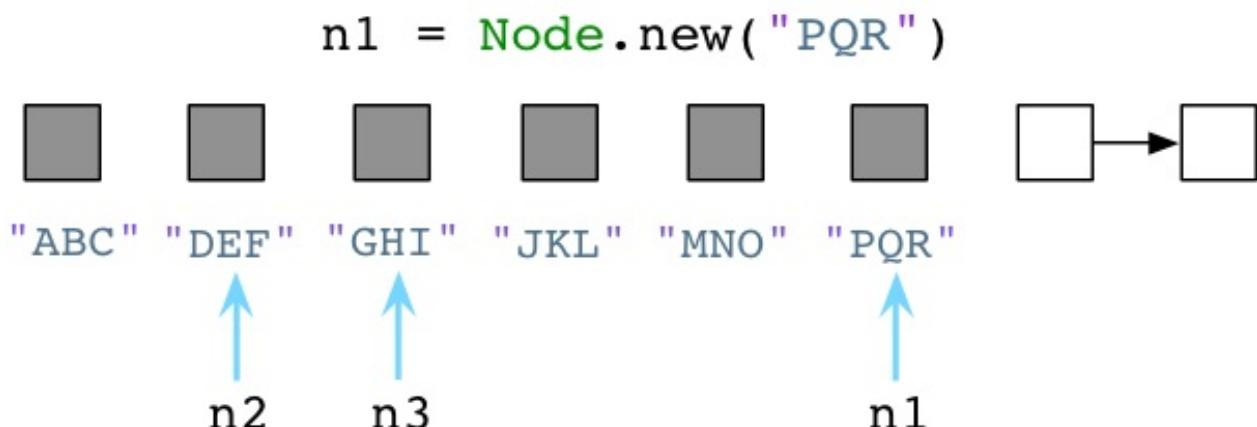
Ruby把无用的对象留在内存里，直到下一次GC执行

回过来看Ruby。随着我们创建越来越多的对象，Ruby会持续寻可用列表里取预创建对象给我们。因此，可用列表会逐渐变短：

```
n1 = Node.new("JKL")
```



...然后更短：



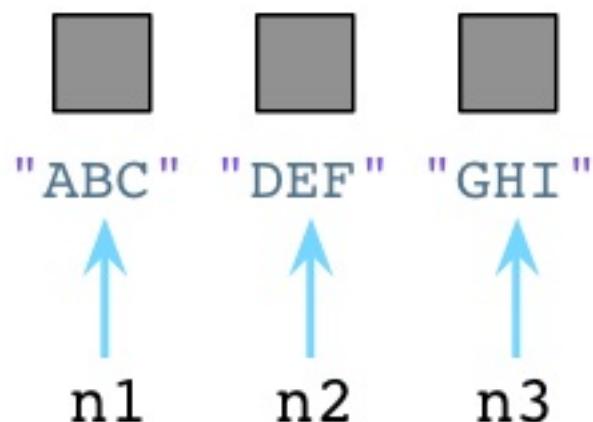
请注意我一直在为变量`n1`赋新值，Ruby把旧值留在原处。`"ABC"`,`"JKL"`和`"MNO"`三个Node实例还滞留在内存中。Ruby不会立即清除代码中不再使用的旧对象！Ruby开发者们就像是住在一间凌乱的房间，地板上摞着衣服，要么洗碗池里都是脏盘子。作为一个Ruby程序员，无用的垃圾对象会一直环绕着你。

2.6 Python 开发者住在卫生之家庭

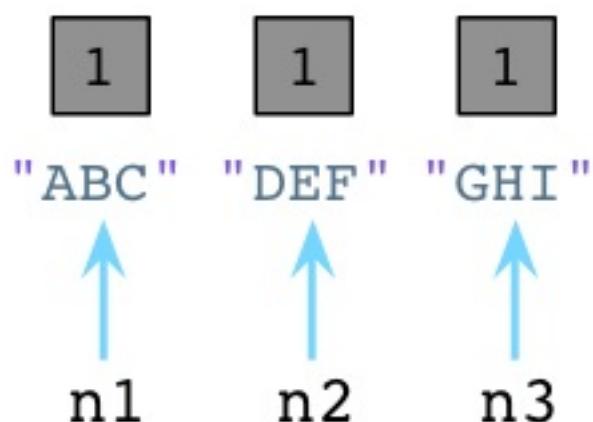


用完的垃圾对象会立即被Python打扫干净

Python与Ruby的垃圾回收机制颇为不同。让我们回到前面提到的三个Python Node对象：

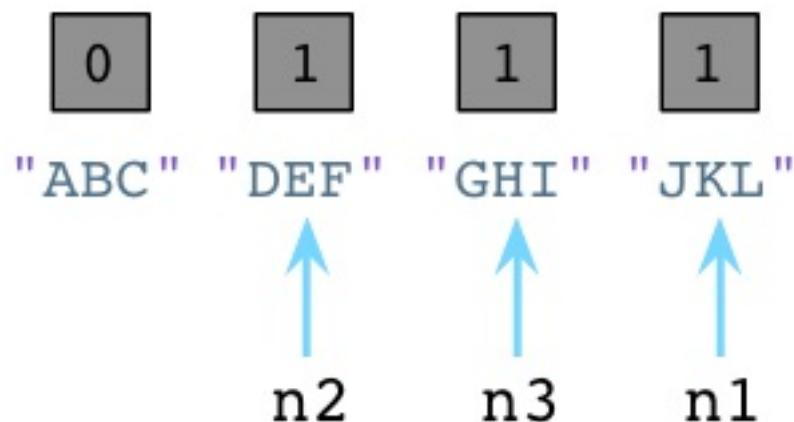


在内部，创建一个对象时，Python总是在对象的C结构体里保存一个整数，称为 `引用数`。期初，Python将这个值设置为1：



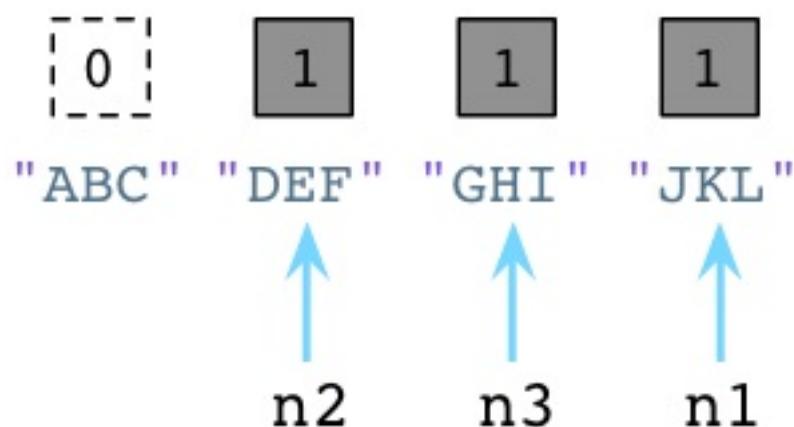
值为1说明分别有个一个指针指向或是引用这三个对象。假如我们现在创建一个新的Node实例，JKL：

```
n1 = Node( "JKL" )
```



与之前一样，Python设置JKL的引用数为1。然而，请注意由于我们改变了n1指向了JKL，不再指向ABC，Python就把ABC的引用数置为0了。此刻，Python垃圾回收器立刻挺身而出！每当对象的引用数减为0，Python立即将其释放，把内存还给操作系统：

```
n1 = Node( "JKL" )
```



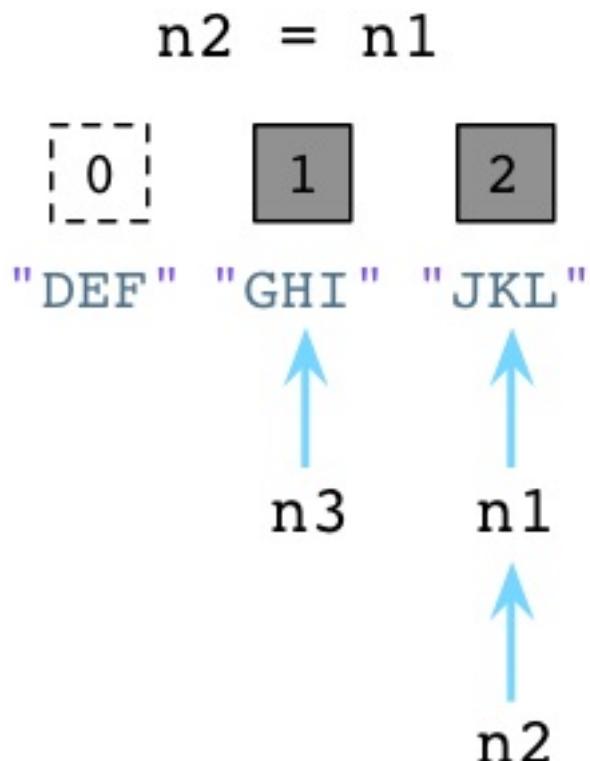
上面Python回收了ABC Node实例使用的内存。记住，Ruby弃旧对象原地于不顾，也不释放它们的内存。

Python的这种垃圾回收算法被称为引用计数。是George-Collins在1960年发明的，恰巧与John McCarthy发明的可用列表算法在同一年出现。就像Mike-Bernstein在6月份哥谭市Ruby大会杰出的垃圾回收机制演讲中说的：“1960年

是垃圾收集器的黄金年代..."

Python开发者工作在卫生之家,你可以想象,有个患有轻度OCD(一种强迫症)的室友一刻不停地跟在你身后打扫,你一放下脏碟子或杯子,有个家伙已经准备好把它放进洗碗机了!

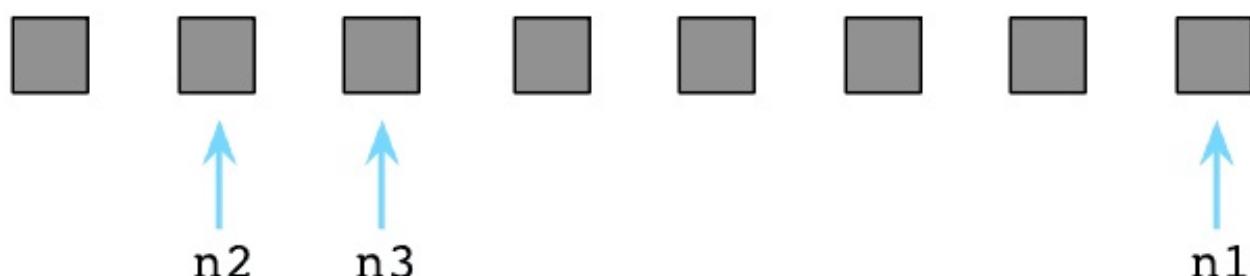
现在来看第二例子。加入我们让n2引用n1:



上图中左边的DEF的引用数已经被Python减少了, 垃圾回收器会立即回收DEF实例。同时JKL的引用数已经变为了2, 因为n1和n2都指向它。

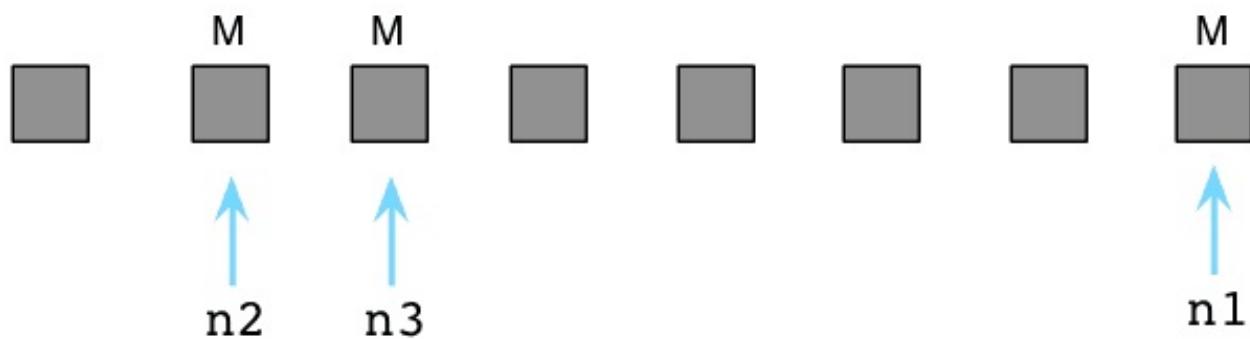
2.7 标记-清除

最终那间凌乱的房间充斥着垃圾, 再不能岁月静好了。在Ruby程序运行了一阵子以后, 可用列表最终被用光光了:

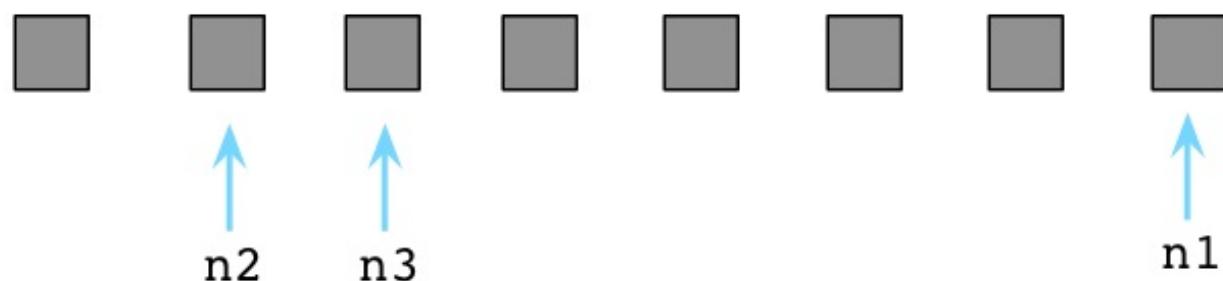


此刻所有Ruby预创建对象都被程序用过了(它们都变灰了), 可用列表里空空如也 (没有白格子了)。

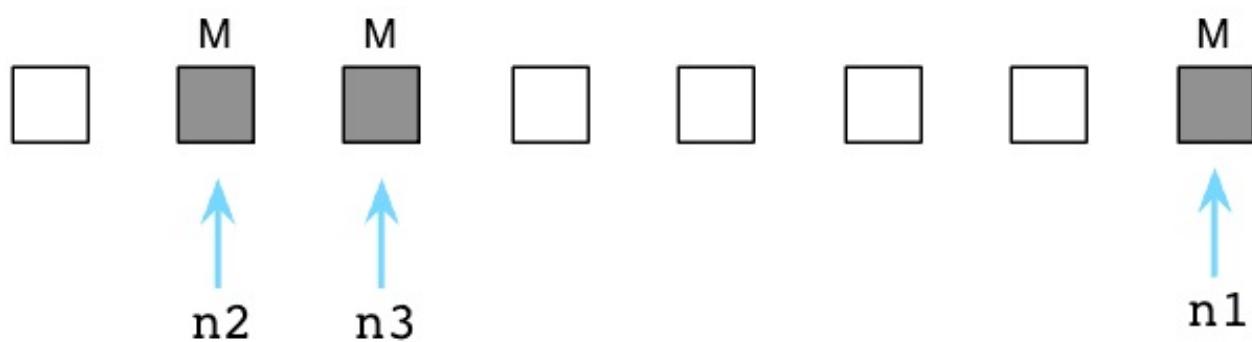
此刻Ruby祭出另一McCarthy发明的算法, 名曰: 标记-清除。首先Ruby把程序停下来, Ruby用"地球停转垃圾回收大法"。之后Ruby轮询所有指针, 变量和代码产生别的引用对象和其他值。同时Ruby通过自身的虚拟机便利内部指针。标记出这些指针引用的每个对象。我在图中使用M表示。



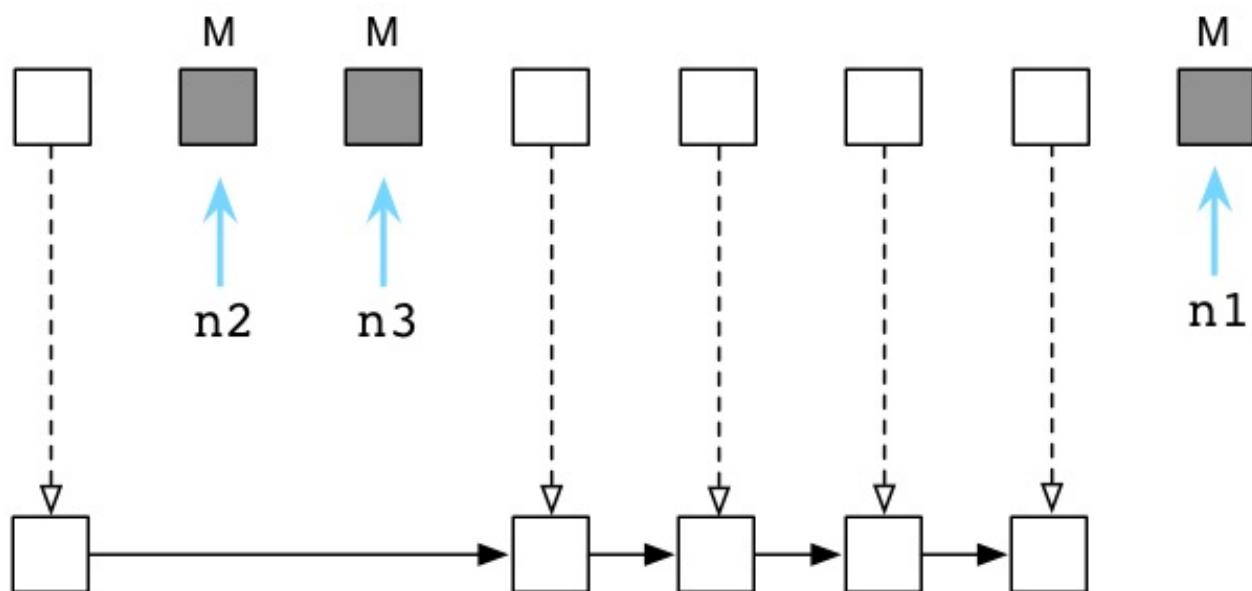
上图中那三个被标M的对象是程序还在使用的。在内部, Ruby实际上使用一串位值, 被称为:可用位图(译注: 还记得《编程珠玑》里的为突发排序吗, 这对离散度不高的有限整数集合具有很强的压缩效果, 用以节约机器的资源。), 来跟踪对象是否被标记了。



如果说被标记的对象是存活的, 剩下的未被标记的对象只能是垃圾, 这意味着我们的代码不再会使用它了。我会在下图中用白格子表示垃圾对象:



接下来Ruby清除这些无用的垃圾对象，把它们送回到可用列表中：



在内部这一切发生得迅雷不及掩耳，因为Ruby实际上不会吧对象从这拷贝到那。而是通过调整内部指针，将其指向一个新链表的方式，来将垃圾对象归位到可用列表中的。

现在等到下回再创建对象的时候Ruby又可以把这些垃圾对象分给我们使用了。在Ruby里，对象们六道轮回，转世投胎，享受多次人生。

2.8 标记-删除 vs. 引用计数

乍一看，Python的GC算法貌似远胜于Ruby的：宁舍洁宇而居秽室乎？为什么Ruby宁愿定期强制程序停止运行，也不使用Python的算法呢？

然而，引用计数并不像第一眼看上去那样简单。有许多原因使得不许多语言不像Python这样使用引用计数GC算法：

首先，它不好实现。Python不得不在每个对象内部留一些空间来处理引用数。这样付出了一小点儿空间上的代价。但更糟糕的是，每个简单的操作（像修改变量或引用）都会变成一个更复杂的操作，因为Python需要增加一个计数，减少另一个，还可能释放对象。

第二点，它相对较慢。虽然Python随着程序执行GC很稳健（一把脏碟子放在洗碗盆里就开始洗啦），但这并不一定更快。Python不停地更新着众多引用数值。特别是当你不再使用一个大数据结构的时候，比如一个包含很多元素的列表，Python可能必须一次性释放大量对象。减少引用数就成了一项复杂的递归过程了。

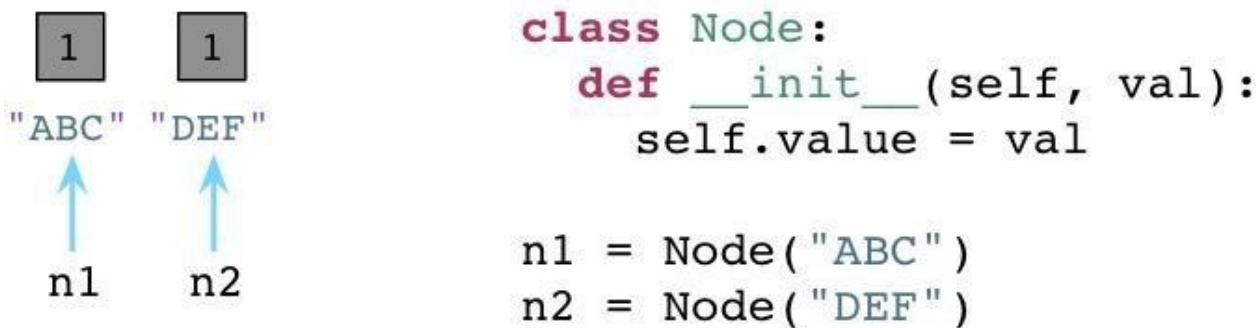
最后，它不是总奏效的。引用计数不能处理环形数据结构--也就是含有循环引用的数据结构。

3. Python中的循环数据结构以及引用计数

3.1 循环引用

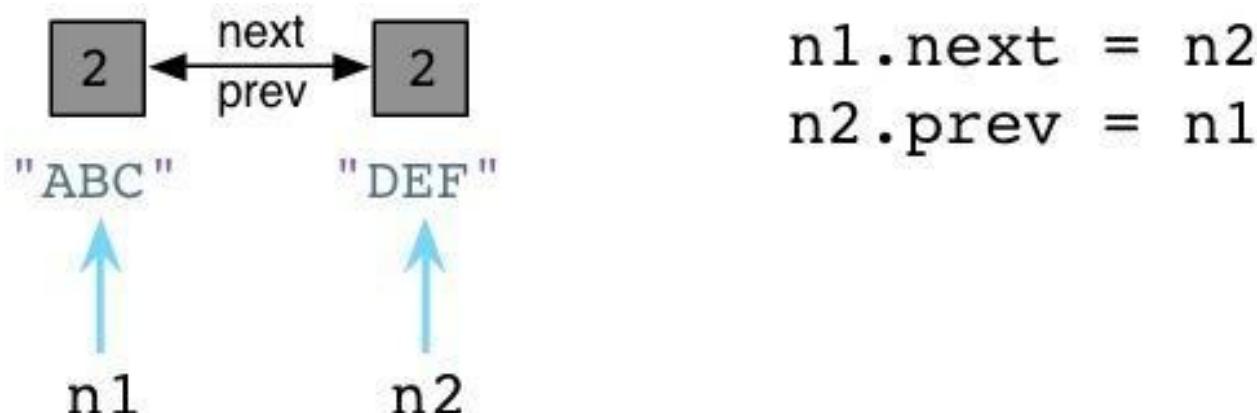
通过上篇，我们知道在Python中，每个对象都保存了一个称为引用计数的整数值，来追踪到底有多少引用指向了这个对象。无论何时，如果我们程序中的一个变量或其他对象引用了目标对象，Python将会增加这个计数值，而当程序停止使用这个对象，则Python会减少这个计数值。一旦计数值被减到零，Python将会释放这个对象以及回收相关内存空间。

从六十年代开始，计算机科学界就面临了一个严重的理论问题，那就是针对引用计数这种算法来说，如果一个数据结构引用了它自身，即如果这个数据结构是一个循环数据结构，那么某些引用计数值是肯定无法变成零的。为了更好地理解这个问题，让我们举个例子。下面的代码展示了一些上周我们所用到的节点类：



我们有一个"构造器"(在Python中叫做`__init__`)，在一个实例变量中存储一个单独的属性。在类定义之后我们创建两个节点，ABC以及DEF，在图中为左边的矩形框。两个节点的引用计数都被初始化为1，因为各有两个引用指向各个节点(n1和n2)。

现在，让我们在节点中定义两个附加的属性，`next`以及`prev`:



跟Ruby不同的是，Python中你可以在代码运行的时候动态定义实例变量或对象属性。这看起来似乎有点像Ruby缺失了某些有趣的魔法。(声明下我不是一个Python程序员，所以可能会存在一些命名方面的错误)。我们设置`n1.next`指向`n2`，同时设置`n2.prev`指向`n1`。现在，我们的两个节点使用循环引用的方式构成了一个 双向链表。同时请注意 ABC 以及 DEF 的引用计数值已经增加到了2。这里有两个指针指向了每个节点：首先是`n1`以及`n2`，其次就是`next`以及`prev`。

现在，假定我们的程序不再使用这两个节点了，我们将`n1`和`n2`都设置为`null`(Python中是`None`)。

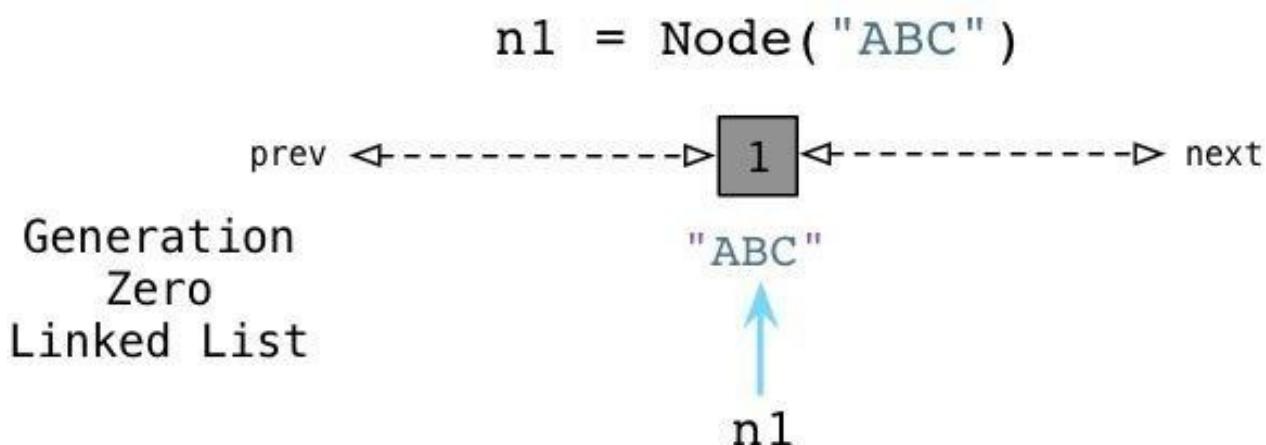


好了，Python会像往常一样将每个节点的引用计数减少到1。

3.2 在Python中的零代(Generation Zero)

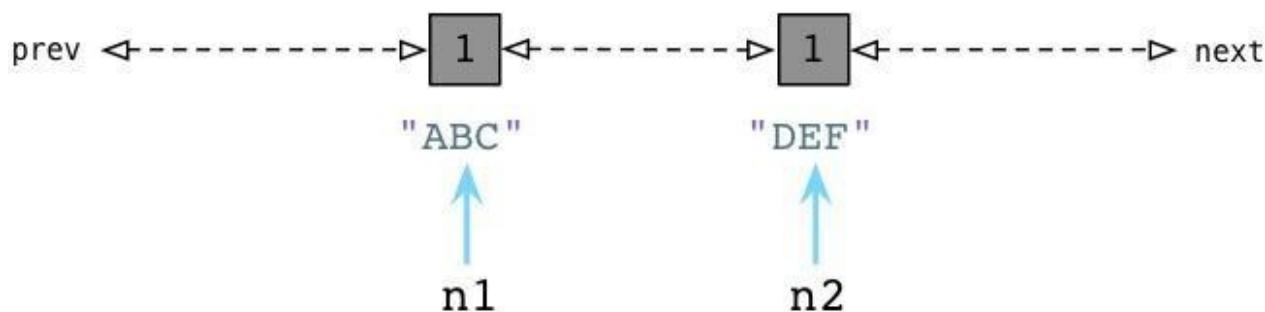
请注意在以上刚刚说到的例子中，我们以一个不是很常见的情况结尾：我们有一个“孤岛”或是一组未使用的、互相指向的对象，但是谁都没有外部引用。换句话说，我们的程序不再使用这些节点对象了，所以我们希望Python的垃圾回收机制能够足够智能去释放这些对象并回收它们占用的内存空间。但是这不可能，因为所有的引用计数都是1而不是0。Python的引用计数算法不能够处理互相指向自己的对象。

这就是为什么Python要引入 Generational GC 算法的原因！正如Ruby使用一个链表(free list)来持续追踪未使用的、自由的对象一样，Python使用一种不同的链表来持续追踪活跃的对象。而不将其称之为“活跃列表”，Python的内部C代码将其称为零代(Generation Zero)。每次当你创建一个对象或其他什么值的时候，Python会将其加入零代链表：



从上边可以看到当我们创建ABC节点的时候，Python将其加入零代链表。请注意到这并不是一个真正的列表，并不能直接在你的代码中访问，事实上这个链表是一个完全内部的Python运行时。相似的，当我们创建DEF节点的时候，Python将其加入同样的链表：

```
n2 = Node("DEF")
```

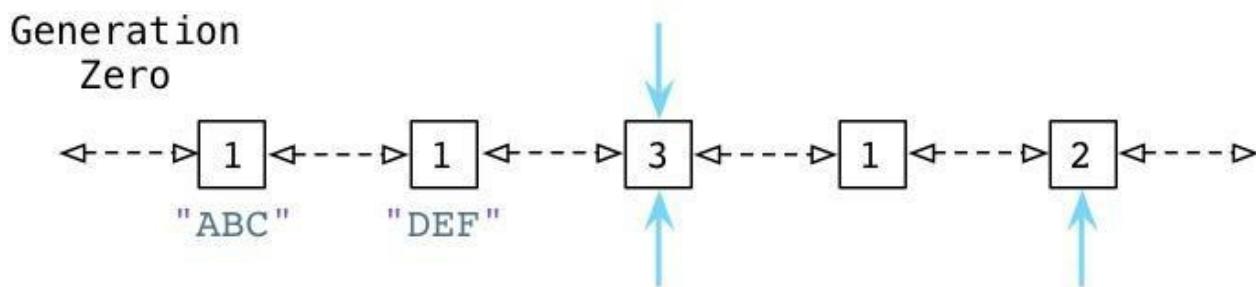


现在零代包含了两个节点对象。(他还将包含Python创建的每个其他值，与一些Python自己使用的内部值。)

3.3 检测循环引用

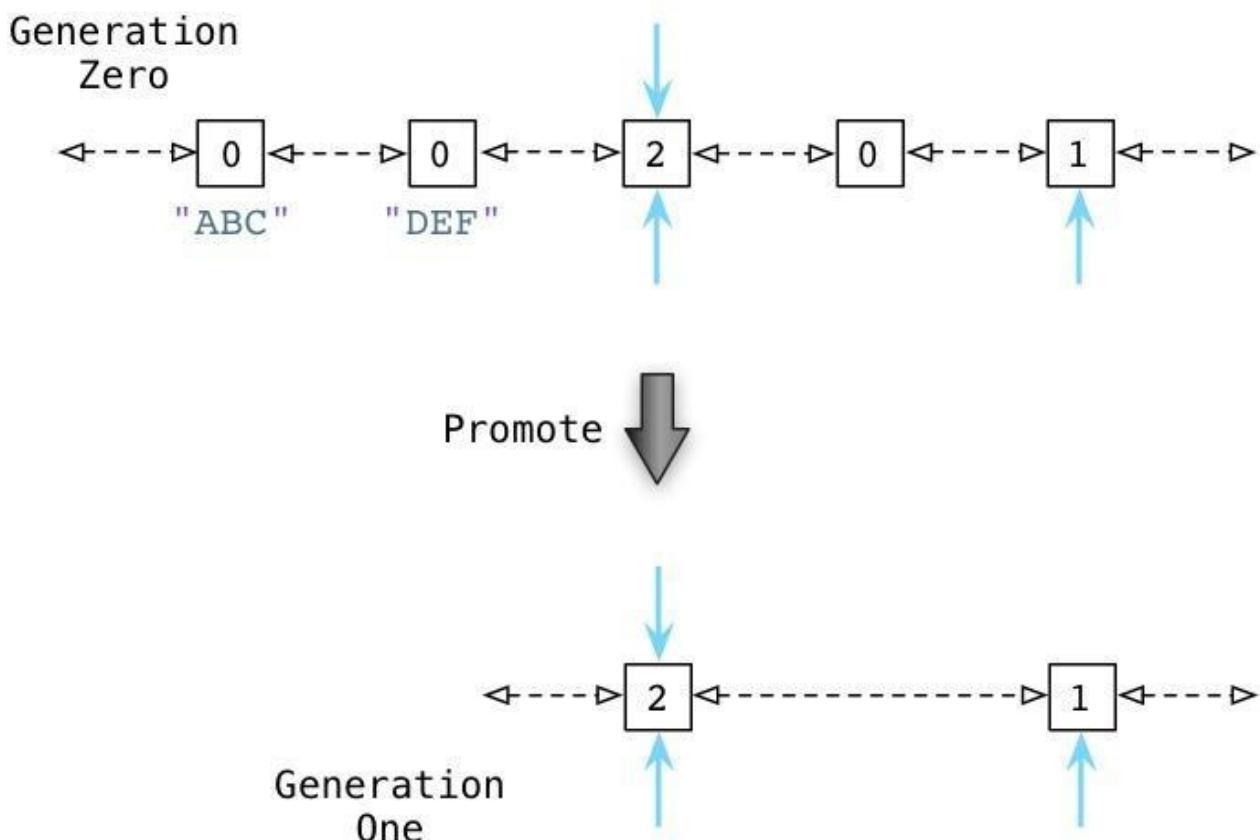
随后，Python会循环遍历零代列表上的每个对象，检查列表中每个互相引用的对象，根据规则减掉其引用计数。在这个过程中，Python会一个接一个的统计内部引用的数量以防过早地释放对象。

为了便于理解，来看一个例子：



从上面可以看到 ABC 和 DEF 节点包含的引用数为1.有三个其他的对象同时存在于零代链表中，蓝色的箭头指示了有一些对象正在被零代链表之外的其他对象所引用。(接下来我们会看到，Python中同时存在另外两个分别被称为一代和二代的链表)。这些对象有着更高的引用计数因为它们正在被其他指针所指向着。

接下来你会看到Python的GC是如何处理零代链表的。



通过识别内部引用，Python能够减少许多零代链表对象的引用计数。在上图的第一行中你能够看见ABC和DEF的引用计数已经变为零了，这意味着收集器可以释放它们并回收内存空间了。剩下的活跃的对象则被移动到一个新的链表：一代链表。

从某种意义上说，Python的GC算法类似于Ruby所用的标记回收算法。周期性地从一个对象到另一个对象追踪引用以确定对象是否还是活跃的，正在被程序所使用的，这正类似于Ruby的标记过程。

Python中的GC阈值

Python什么时候会进行这个标记过程？随着你的程序运行，Python解释器保持对新创建的对象，以及因为引用计数为零而被释放掉的对象的追踪。从理论上说，这两个值应该保持一致，因为程序新建的每个对象都应该最终被释放掉。

当然，事实并非如此。因为循环引用的原因，并且因为你的程序使用了一些比其他对象存在时间更长的对象，从而被分配对象的计数值与被释放对象的计数值之间的差异在逐渐增长。一旦这个差异累计超过某个阈值，则Python

的收集机制就启动了，并且触发上边所说到的零代算法，释放“浮动的垃圾”，并且将剩下的对象移动到一代列表。

随着时间的推移，程序所使用的对象逐渐从零代列表移动到一代列表。而Python对于一代列表中对象的处理遵循同样的方法，一旦被分配计数值与被释放计数值累计到达一定阈值，Python会将剩下的活跃对象移动到二代列表。

通过这种方法，你的代码所长期使用的对象，那些你的代码持续访问的活跃对象，会从零代链表转移到一代再转移到二代。通过不同的阈值设置，Python可以在不同的时间间隔处理这些对象。Python处理零代最为频繁，其次是一代然后才是二代。

弱代假说

来看看代垃圾回收算法的核心行为：垃圾回收器会更频繁的处理新对象。一个新的对象即是你的程序刚刚创建的，而一个来的对象则是经过了几个时间周期之后仍然存在的对象。Python会在当一个对象从零代移动到一代，或是从一代移动到二代的过程中提升(promote)这个对象。

为什么要这么做？这种算法的根源来自于弱代假说(weak generational hypothesis)。这个假说由两个观点构成：首先是年亲的对象通常死得也快，而老对象则很有可能存活更长的时间。

假定现在我用Python或是Ruby创建一个新对象：

```
n1 = Node("ABC")      n1 = Node.new("ABC")
```

根据假说，我的代码很可能仅仅会使用ABC很短的时间。这个对象也许仅仅只是一个方法中的中间结果，并且随着方法的返回这个对象就将变成垃圾了。大部分的新对象都是如此般地很快变成垃圾。然而，偶尔程序会创建一些很重要的，存活时间比较长的对象-例如web应用中的session变量或是配置项。

通过频繁的处理零代链表中的新对象，Python的垃圾收集器将把时间花在更有意义的地方：它处理那些很快就可能变成垃圾的新对象。同时只在很少的时候，当满足阈值的条件，收集器才回去处理那些老变量。

垃圾回收(三)-gc模块

一. 垃圾回收机制

Python中的垃圾回收是以引用计数为主，分代收集为辅。

1、导致引用计数+1的情况

- 对象被创建，例如`a=23`
- 对象被引用，例如`b=a`
- 对象被作为参数，传入到一个函数中，例如`func(a)`
- 对象作为一个元素，存储在容器中，例如`list1=[a, a]`

2、导致引用计数-1的情况

- 对象的别名被显式销毁，例如`del a`
- 对象的别名被赋予新的对象，例如`a=24`
- 一个对象离开它的作用域，例如`f`函数执行完毕时，`func`函数中的局部变量（全局变量不会）
- 对象所在的容器被销毁，或从容器中删除对象

3、查看一个对象的引用计数

```
import sys
a = "hello world"
sys.getrefcount(a)
```

可以查看`a`对象的引用计数，但是比正常计数大1，因为调用函数的时候传入`a`，这会让`a`的引用计数+1

二.循环引用导致内存泄露

引用计数的缺陷是循环引用的问题

```
import gc

class ClassA():
    def __init__(self):
        print('object born, id:%s' % str(hex(id(self))))


def f2():
    while True:
        c1 = ClassA()
        c2 = ClassA()
        c1.t = c2
        c2.t = c1
        del c1
        del c2

#把python的gc关闭
gc.disable()

f2()
```

执行f2(), 进程占用的内存会不断增大。

- 创建了c1, c2后这两块内存的引用计数都是1, 执行 c1.t=c2 和 c2.t=c1 后, 这两块内存的引用计数变成2.
- 在del c1后, 内存1的对象的引用计数变为1, 由于不是为0, 所以内存1的对象不会被销毁, 所以内存2的对象的引用数依然是2, 在del c2后, 同理, 内存1的对象, 内存2的对象的引用数都是1。
- 虽然它们两个的对象都是可以被销毁的, 但是由于循环引用, 导致垃圾回收器都不会回收它们, 所以就会导致内存泄露。

三. 垃圾回收

```
#coding=utf-8
import gc

class ClassA():
    def __init__(self):
        print('object born,id:%s'%str(hex(id(self))))
    # def __del__(self):
    #     print('object del,id:%s'%str(hex(id(self)))))

def f3():
    print("-----0-----")
    # print(gc.collect())
    c1 = ClassA()
    c2 = ClassA()
    c1.t = c2
    c2.t = c1
    print("-----1-----")
    del c1
    del c2
    print("-----2-----")
    print(gc.garbage)
    print("-----3-----")
    print(gc.collect()) #显式执行垃圾回收
    print("-----4-----")
    print(gc.garbage)
    print("-----5-----")

if __name__ == '__main__':
    gc.set_debug(gc.DEBUG_LEAK) #设置gc模块的日志
    f3()
```

python2运行结果：

```
-----0-----
```

```
object born, id:0x724b20
object born, id:0x724b48
-----1-----
-----2-----
[]
-----3-----
gc: collectable <ClassA instance at 0x724b20>
gc: collectable <ClassA instance at 0x724b48>
gc: collectable <dict 0x723300>
gc: collectable <dict 0x71bf60>
4
-----4-----
[<__main__.ClassA instance at 0x724b20>, <__main__.ClassA instar
-----5-----
```

说明:

- 垃圾回收后的对象会放在`gc.garbage`列表里面
- `gc.collect()`会返回不可达的对象数目，4等于两个对象以及它们对应的`dict`

有三种情况会触发垃圾回收:

1. 调用`gc.collect()`,
2. 当`gc`模块的计数器达到阈值的时候。
3. 程序退出的时候

四.gc模块常用功能解析

`gc`模块提供一个接口给开发者设置垃圾回收的选项。上面说到，采用引用计数的方法管理内存的一个缺陷是循环引用，而`gc`模块的一个主要功能就是解决循环引用的问题。

常用函数:

- 1、`gc.set_debug(flags)` 设置gc的debug日志，一般设置为`gc.DEBUG_LEAK`
- 2、`gc.collect([generation])` 显式进行垃圾回收，可以输入参数，0代表只检查第一代的对象，1代表检查一，二代的对象，2代表检查一，二，三代的对象，如果不传参数，执行一个full collection，也就是等于传2。返回不可达(unreachable objects) 对象的数目
- 3、`gc.get_threshold()` 获取的gc模块中自动执行垃圾回收的频率。
- 4、`gc.set_threshold(threshold0[, threshold1[, threshold2]])` 设置自动执行垃圾回收的频率。
- 5、`gc.get_count()` 获取当前自动执行垃圾回收的计数器，返回一个长度为3的列表

gc模块的自动垃圾回收机制

必须要import gc模块，并且 `is_enable()=True` 才会启动自动垃圾回收。

这个机制的主要作用就是发现并处理不可达的垃圾对象。

垃圾回收=垃圾检查+垃圾回收

在Python中，采用分代收集的方法。把对象分为三代，一开始，对象在创建的时候，放在一代中，如果在一次一代的垃圾检查中，该对象存活下来，就会被放到二代中，同理在一次二代的垃圾检查中，该对象存活下来，就会被放到三代中。

gc模块里面会有一个长度为3的列表的计数器，可以通过`gc.get_count()`获取。

例如(488,3,0)，其中488是指距离上一次一代垃圾检查，Python分配内存的数目减去释放内存的数目，注意是内存分配，而不是引用计数的增加。例如：

```
print gc.get_count() # (590, 8, 0)
a = ClassA()
print gc.get_count() # (591, 8, 0)
```

```
del a
print gc.get_count() # (590, 8, 0)
```

3是指距离上一次二代垃圾检查，一代垃圾检查的次数，同理，0是指距离上一次三代垃圾检查，二代垃圾检查的次数。

gc模块快有一个自动垃圾回收的 阈值，即通过gc.get_threshold函数获取到的长度为3的元组，例如(700,10,10)每一次计数器的增加，gc模块就会检查增加后的计数是否达到阈值的数目，如果是，就会执行对应的代数的垃圾检查，然后重置计数器

例如，假设阈值是(700,10,10):

当计数器从(699, 3, 0)增加到(700, 3, 0)，gc模块就会执行gc.collect(0)，即检查
 当计数器从(699, 9, 0)增加到(700, 9, 0)，gc模块就会执行gc.collect(1)，即检查
 当计数器从(699, 9, 9)增加到(700, 9, 9)，gc模块就会执行gc.collect(2)，即检查

注意点

gc模块唯一处理不了的是循环引用的类都有__del__方法，所以项目中要避免定义__del__方法

```
import gc

class ClassA():
    pass
    # def __del__(self):
    #     print('object born, id:%s' % str(hex(id(self))))


gc.set_debug(gc.DEBUG_LEAK)
a = ClassA()
b = ClassA()

a.next = b
```

```
b.prev = a

print "--1--"
print gc.collect()
print "--2--"
del a
print "--3--"
del b
print "--3-1--"
print gc.collect()
print "--4--"
```

运行结果：

```
--1--
0
--2--
--3--
--3-1--
gc: collectable <ClassA instance at 0x21248c8>
gc: collectable <ClassA instance at 0x21248f0>
gc: collectable <dict 0x2123030>
gc: collectable <dict 0x2123150>
4
--4--
```

如果把**del**打开，运行结果为：

```
--1--
0
--2--
--3--
--3-1--
gc: uncollectable <ClassA instance at 0x6269b8>
gc: uncollectable <ClassA instance at 0x6269e0>
gc: uncollectable <dict 0x61bed0>
```

```
gc: uncollectable <dict 0x6230c0>
4
--4--
```

内建属性

```
"teachclass.py"

class Person(object):
    pass
```

python3.5中类的内建属性和方法

```
In [1]: from teachclass import Person
```

```
In [2]: dir(Person)
```

```
Out[2]:
```

```
['__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__']
```

经典类(旧式类),早期如果没有要继承的父类,继承里空着不写的类

```
#py2中无继承父类, 称之经典类, py3中已默认继承object
```

```
class Person:  
    pass
```

子类没有实现 `__init__` 方法时， 默认自动调用父类的。如定义 `__init__` 方法时， 需自己手动调用父类的 `__init__` 方法

常用专有属性	说明	触发方式
<code>__init__</code>	构造初始化函数	创建实例后,赋值时使用,在 <code>__new__</code> 后
<code>__new__</code>	生成实例所需属性	创建实例时
<code>__class__</code>	实例所在的类	实例. <code>__class__</code>
<code>__str__</code>	实例字符串表示,可读性	print(类实例),如没实现,使用repr结果
<code>__repr__</code>	实例字符串表示,准确性	类实例 回车 或者 print(repr(类实例))
<code>__del__</code>	析构	<code>del</code> 删除实例
<code>__dict__</code>	实例自定义属性	<code>vars(实例.__dict__)</code>
<code>__doc__</code>	类文档,子类不继承	<code>help(类或实例)</code>
<code>__getattribute__</code>	属性访问拦截器	访问实例属性时
<code>__bases__</code>	类的所有父类构成元素	类名. <code>__bases__</code>

`__getattribute__` 例子:

```
class Itcast(object):  
    def __init__(self, subject1):  
        self.subject1 = subject1  
        self.subject2 = 'cpp'
```

```
#属性访问时拦截器，打log
def __getattribute__(self,obj):
    if obj == 'subject1':
        print('log subject1')
        return 'redirect python'
    else:    #测试时注释掉这2行，将找不到subject2
        return object.__getattribute__(self,obj)

def show(self):
    print('this is Itcast')

s = Itcast("python")
print(s.subject1)
print(s.subject2)
```

运行结果：

```
log subject1
redirect python
cpp
```

__getattribute__的坑

```
class Person(object):
    def __getattribute__(self,obj):
        print("---test---")
        if obj.startswith("a"):
            return "hahha"
        else:
            return self.test

    def test(self):
        print("heihei")
```

```
t.Person()  
  
t.a #返回hahha  
  
t.b #会让程序死掉  
#原因是：当t.b执行时，会调用Person类中定义的__getattribute__方  
#if条件不满足，所以 程序执行else里面的代码，即return self.test  
#self.test的值返回，那么首先要获取self.test的值，因为self此时就  
#t.test 此时要获取t这个对象的test属性，那么就会跳转到__getattr__  
#生了递归调用，由于这个递归过程中 没有判断什么时候推出，所以这个程  
#每次调用函数，就需要保存一些数据，那么随着调用的次数越来越多，最终  
#  
# 注意：以后不要在__getattribute__方法中调用self.xxxx
```

内建函数

Build-in Function, 启动python解释器， 输入 `dir(__builtins__)`，可以看到很多python解释器启动后默认加载的属性和函数，这些函数称之为内建函数，这些函数因为在编程时使用较多， cpython解释器用c语言实现了这些函数，启动解释器时默认加载。

这些函数数量众多，不宜记忆，开发时不是都用到的，待用到时再 `help(function)`，查看如何使用，或结合百度查询即可，在这里介绍些常用的内建函数。

range

```
range(stop) -> list of integers
range(start, stop[, step]) -> list of integers
```

- start:计数从start开始。默认是从0开始。例如`range (5)` 等价于`range (0, 5)`；
- stop:到stop结束，但不包括stop.例如：`range (0, 5)` 是[0, 1, 2, 3, 4] 没有5
- step:每次跳跃的间距，默认为1。例如：`range (0, 5)` 等价于`range(0, 5, 1)`

python2中range返回列表， python3中range返回一个迭代值。如果想得到列表，可通过list函数

```
a = range(5)
list(a)
```

创建列表的另外一种方法

```
In [21]: testList = [x+2 for x in range(5)]
```

```
In [22]: testList
Out[22]: [2, 3, 4, 5, 6]
```

map函数

map函数会根据提供的函数对指定序列做映射

```
map( ... )
    map(function, sequence[, sequence, ...]) -> list
```

- function:是一个函数
- sequence:是一个或多个序列,取决于function需要几个参数
- 返回值是一个list

参数序列中的每一个元素分别调用function函数, 返回包含每次function函数返回值的list。

```
#函数需要一个参数
map(lambda x: x*x, [1, 2, 3])
#结果为:[1, 4, 9]

#函数需要两个参数
map(lambda x, y: x+y, [1, 2, 3], [4, 5, 6])
#结果为:[5, 7, 9]

def f1( x, y ):
    return (x,y)

l1 = [ 0, 1, 2, 3, 4, 5, 6 ]
l2 = [ 'Sun', 'M', 'T', 'W', 'T', 'F', 'S' ]
l3 = map( f1, l1, l2 )
print(list(l3))
#结果为:[(0, 'Sun'), (1, 'M'), (2, 'T'), (3, 'W'), (4, 'T'), (5,
```

filter函数

filter函数会对指定序列执行过滤操作

```
filter(...)
```

```
filter(function or None, sequence) -> list, tuple, or string
```

Return those items of sequence for which function(item) is true. If function is None, return the items that are true. If sequence is a string, return the same type, else return a list.

- **function:**接受一个参数，返回布尔值True或False
- **sequence:**序列可以是str, tuple, list

filter函数会对序列参数sequence中的每个元素调用function函数，最后返回的结果包含调用结果为True的元素。

返回值的类型和参数sequence的类型相同

```
filter(lambda x: x%2, [1, 2, 3, 4])
[1, 3]
```

```
filter(None, "she")
'she'
```

reduce函数

reduce函数，reduce函数会对参数序列中元素进行累积

```
reduce(...)
```

```
reduce(function, sequence[, initial]) -> value
```

Apply a function of two arguments cumulatively to the items from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). If initial is present, it is placed before the first call to function. For example, reduce(lambda x, y: x+y, [1, 2, 3], -1) calculates -1+(1+2)+3=5.

of the sequence in the calculation, and serves as a default sequence is empty.

- **function**:该函数有两个参数
- **sequence**:序列可以是str, tuple, list
- **initial**:固定初始值

reduce依次从sequence中取一个元素，和上一次调用function的结果做参数再次调用function。第一次调用function时，如果提供initial参数，会以sequence中的第一个元素和initial作为参数调用function，否则会以序列sequence中的前两个元素做参数调用function。注意function函数不能为None。

```
reduce(lambda x, y: x+y, [1, 2, 3, 4])  
10  
  
reduce(lambda x, y: x+y, [1, 2, 3, 4], 5)  
15  
  
reduce(lambda x, y: x+y, ['aa', 'bb', 'cc'], 'dd')  
'ddaabbcc'
```

在Python3里,reduce函数已经被从全局名字空间里移除了,它现在被放置在fucntools模块里用的话要先引入: `from functools import reduce`

sorted函数

```
sorted(...)  
sorted(iterable, cmp=None, key=None, reverse=False) --> new
```

```
In [14]: sorted([1,4,2,6,3,5])
Out[14]: [1, 2, 3, 4, 5, 6]

In [15]: sorted([1,4,2,6,3,5],reverse=1)
Out[15]: [6, 5, 4, 3, 2, 1]

In [16]: sorted(['dd','aa','cc','bb'])
Out[16]: ['aa', 'bb', 'cc', 'dd']

In [17]: sorted(['dd','aa','cc','bb'],reverse=1)
Out[17]: ['dd', 'cc', 'bb', 'aa']
```

集合set

集合与之前列表、元组类似，可以存储多个数据，但是这些数据是不重复的。集合对象还支持union(联合), intersection(交), difference(差)和symmetric_difference(对称差集)等数学运算。

```
>>> x = set('abcd')
>>> x
{'c', 'a', 'b', 'd'}
>>> type(x)
<class 'set'>
>>>
>>>
>>> y = set(['h', 'e', 'l', 'l', 'o'])
>>> y
{'h', 'e', 'o', 'l'}
>>>
>>>
>>> z = set('spam')
>>> z
{'s', 'a', 'm', 'p'}
>>>
>>>
>>> y&z #交集
set()
>>>
>>>
>>> x&z #交集
{'a'}
>>>
>>>
>>> x|y #并集
{'a', 'e', 'd', 'l', 'c', 'h', 'o', 'b'}
>>>
>>> x-y #差集
```

```
{'c', 'a', 'b', 'd'}  
>>>  
>>>  
>>> x^z #对称差集(在x或z中, 但不会同时出现在二者中)  
{'m', 'd', 's', 'c', 'b', 'p'}  
>>>  
>>>  
>>> len(x)  
4  
>>> len(y)  
4  
>>> len(z)  
4  
>>>
```

functools

functools 是python2.5被引入的,一些工具函数放在此包里。

python2.7中

```
In [2]: dir(functools)
Out[2]:
['WRAPPER_ASSIGNMENTS',
 'WRAPPER_UPDATES',
 '__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 'cmp_to_key',
 'partial',
 'reduce',
 'total_ordering',
 'update_wrapper',
 'wraps']
```

python3.5中

```
import functools
dir(functools)
```

运行结果：

```
['MappingProxyType',
 'RLock',
 'WRAPPER_ASSIGNMENTS',
 'WRAPPER_UPDATES',
 'WeakKeyDictionary',
 '_CacheInfo',
 '_HashedSeq',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__']
```

```
'__loader__',
'__name__',
'__package__',
'__spec__',
'_c3_merge',
'_c3_mro',
'_compose_mro',
'_convert',
'_find_impl',
'_ge_from_gt',
'_ge_from_le',
'_ge_from_lt',
'_gt_from_ge',
'_gt_from_le',
'_gt_from_lt',
'_le_from_ge',
'_le_from_gt',
'_le_from_lt',
'_lru_cache_wrapper',
'_lt_from_ge',
'_lt_from_gt',
'_lt_from_le',
'_make_key',
'cmp_to_key',
'get_cache_token',
'lru_cache',
'namedtuple',
'partial',
'partialmethod',
'reduce',
'singledispatch',
'total_ordering',
'update_wrapper',
'wraps']
```

python3中增加了更多工具函数，做业务开发时大多情况下用不到，此处介绍使用频率较高的2个函数。

partial函数(偏函数)

把一个函数的某些参数设置默认值，返回一个新的函数，调用这个新函数会更简单。

```
import functools

def showarg(*args, **kw):
    print(args)
    print(kw)

p1=functools.partial(showarg, 1, 2, 3)
p1()
p1(4, 5, 6)
p1(a='python', b='itcast')

p2=functools.partial(showarg, a=3, b='linux')
p2()
p2(1, 2)
p2(a='python', b='itcast')
```

```
python@ubuntu:~/workspace/test$ python t6.py
(1, 2, 3)
{}
(1, 2, 3, 4, 5, 6)
{}
(1, 2, 3)
{'a': 'python', 'b': 'itcast'}
()
{'a': 3, 'b': 'linux'}
(1, 2)
{'a': 3, 'b': 'linux'}
()
{'a': 'python', 'b': 'itcast'}
```

wraps函数

使用装饰器时，有一些细节需要被注意。例如，被装饰后的函数其实已经是另外一个函数了（函数名等函数属性会发生改变）。

添加后由于函数名和函数的doc发生了改变，对测试结果有一些影响，例如：

```
def note(func):
    "note function"
    def wrapper():
        "wrapper function"
        print('note something')
        return func()
    return wrapper

@note
def test():
    "test function"
    print('I am test')

test()
print(test.__doc__)
```

运行结果

```
note something
I am test
wrapper function
```

所以，Python的functools包中提供了一个叫wraps的装饰器来消除这样的副作用。例如：

```
import functools
def note(func):
    "note function"
    @functools.wraps(func)
    def wrapper():
        "wrapper function"
        print('note something')
        return func()
    return wrapper

@note
```

```
def test():
    "test function"
    print('I am test')

test()
print(test.__doc__)
```

运行结果

```
note something
I am test
test function
```

模块进阶

Python有一套很有用的标准库(standard library)。标准库会随着Python解释器，一起安装在你的电脑中的。它是Python的一个组成部分。这些标准库是Python为你准备好的利器，可以让编程事半功倍。

常用标准库

标准库	说明
builtins	内建函数默认加载
os	操作系统接口
sys	Python自身的运行环境
functools	常用的工具
json	编码和解码 JSON 对象
logging	记录日志，调试
multiprocessing	多进程
threading	多线程
copy	拷贝
time	时间
datetime	日期和时间
calendar	日历
hashlib	加密算法
random	生成随机数
re	字符串正则匹配
socket	标准的 BSD Sockets API
shutil	文件和目录管理

hashlib

```
import hashlib
m = hashlib.md5()      #创建hash对象, md5:(message-Digest Algorithm)
print m                  #<md5 HASH object>
m.update('itcast')     #更新哈希对象以字符串参数
print m.hexdigest()    #返回十六进制数字字符串
```

应用实例

用于注册、登录....

```
import hashlib
import datetime
KEY_VALUE = 'Itcast'
now = datetime.datetime.now()
m = hashlib.md5()
str = '%s%s' % (KEY_VALUE, now.strftime("%Y%m%d"))
m.update(str.encode('utf-8'))
value = m.hexdigest()
print(value)
```

运行结果:

```
8ad2d682e3529dac50e586fee8dc05c0
```

更多标准库

http://python.usyiyi.cn/translate/python_352/library/index.html

常用扩展库

扩展库	说明
requests	使用的是 urllib3, 继承了urllib2的所有特性
urllib	基于http的高层库
scrapy	爬虫
beautifulsoup4	HTML/XML的解析器
celery	分布式任务调度模块
redis	缓存
Pillow(PIL)	图像处理
xlsxwriter	仅写excel功能,支持xlsx
xlwt	仅写excel功能,支持xls ,2013或更早版office
xlrd	仅读excel功能
elasticsearch	全文搜索引擎
pymysql	数据库连接库
mongoengine/pymongo	mongodbpython接口
matplotlib	画图
numpy/scipy	科学计算
django/tornado/flask	web框架
xmltodict	xml 转 dict
SimpleHTTPServer	简单地HTTP Server,不使用Web框架
gevent	基于协程的Python网络库
fabric	系统管理
pandas	数据处理库
scikit-learn	机器学习库

就可以运行起来静态服务。平时用它预览和下载文件太方便了。

在终端中输入命令：

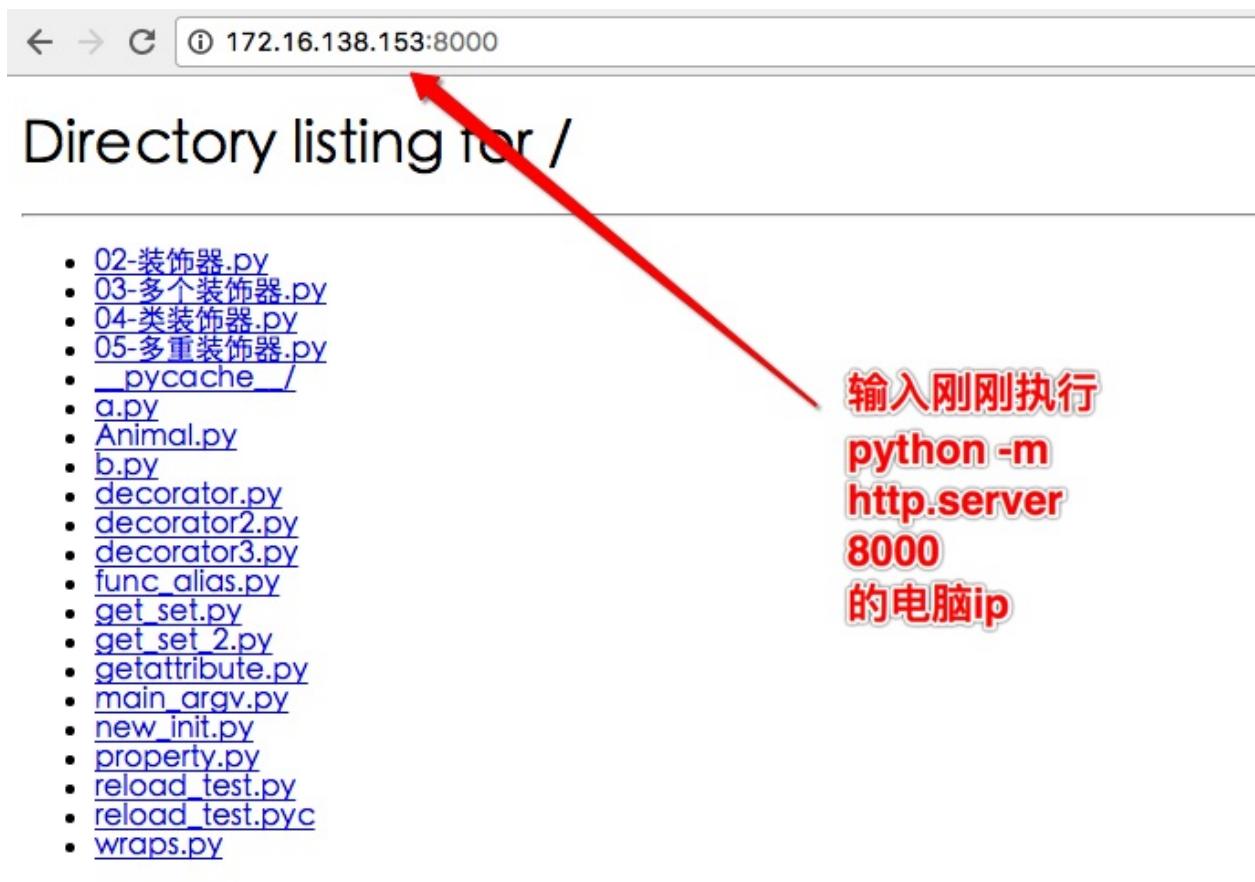
python2中

```
python -m SimpleHTTPServer PORT
```

python3中

```
python -m http.server PORT
```

```
[python@ubuntu:~/Desktop/02-Python就业班/test$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 ...
```



读写excel文件

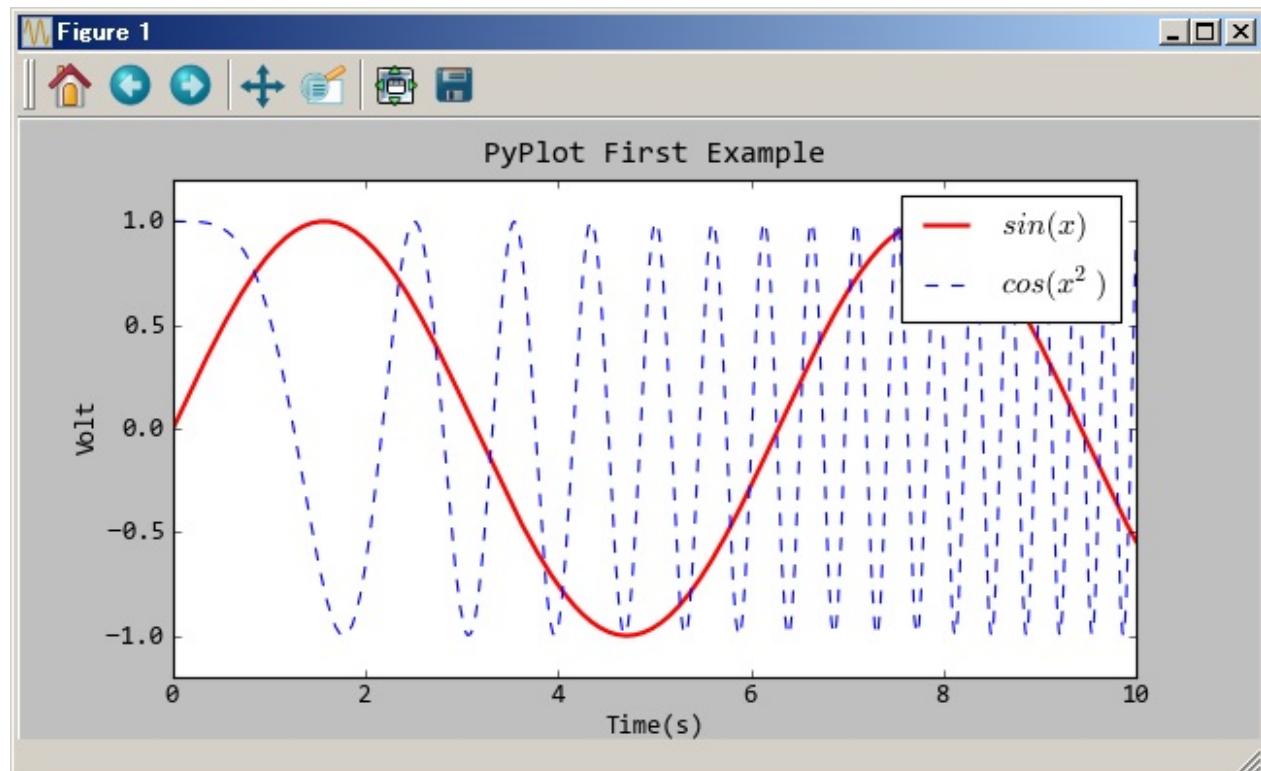
1. 安装个easy_install工具

```
sudo apt-get install python-setuptools
```

2.安装模块

```
sudo easy_install xlrd  
sudo easy_install xlwt
```

matplotlib



调试

pdb

pdb是基于命令行的调试工具，非常类似gnu的gdb（调试c/c++）。

命令	简写命令	作用
break	b	设置断点
continue	c	继续执行程序
list	l	查看当前行的代码段
step	s	进入函数
return	r	执行代码直到从当前函数返回
quit	q	中止并退出
next	n	执行下一行
print	p	打印变量的值
help	h	帮助
args	a	查看传入参数
	回车	重复上一条命令
break	b	显示所有断点
break lineno	b lineno	在指定行设置断点
break file:lineno	b file:lineno	在指定文件的行设置断点
clear num		删除指定断点
bt		查看函数调用栈帧

执行时调试

程序启动，停止在第一行等待单步调试。

```
python -m pdb some.py
```

交互调试

进入python或ipython解释器

```
import pdb  
pdb.run('testfun(args)') #此时会打开pdb调试，注意：先使用s跳转到这个te
```

程序里埋点

当程序执行到pdb.set_trace() 位置时停下来调试

代码上下文

...

```
import pdb  
pdb.set_trace()
```

...

日志调试

print大法好

使用pdb调试的5个demo

demo 1

```
import pdb
a = "aaa"
pdb.set_trace()
b = "bbb"
c = "ccc"
final = a + b + c
print final

#调试方法

# 《1 显示代码》
# 1---->能够显示当前调试过程中的代码，其实1表示list列出的意思
#如下，途中，-> 指向的地方表示要将要执行的位置
# 2      a = "aaa"
# 3      pdb.set_trace()
# 4      b = "bbb"
# 5      c = "ccc"
# 6      pdb.set_trace()
# 7  ->  final = a + b + c
# 8      print final

# 《2 执行下一行代码》
# n---->能够向下执行一行代码，然后停止运行等待继续调试 n表示next的意思

# 《3 查看变量的值》
# p---->能够查看变量的值，p表示print打印输出的意思
#例如：
# p name 表示查看变量name的值
```

demo 2

```
import pdb
a = "aaa"
pdb.set_trace()
b = "bbb"
c = "ccc"
```

```

pdb.set_trace()
final = a + b + c
print final

# 《4 将程序继续运行》
# c----->让程序继续向下执行，与n的区别是n只会执行下面的一行代码，而c会像py
# 《5 set_trace()》
# 如果程序中有多个set_trace()，那么能够让程序在使用c的时候停留在下一个set

```

demo 3

```

#coding=utf-8
import pdb

def combine(s1,s2):
    s3 = s1 + s2 + s1
    s3 = ' ' + s3 + ' '
    return s3

a = "aaa"
pdb.set_trace()
b = "bbb"
c = "ccc"
final = combine(a,b)
print final

# 《6 设置断点》
# b----->设置断点，即当使用c的时候，c可以在遇到set_trace()的时候停止，也可
# 例如：
#b 11 在第11行设置断点，注意这个11可以使用l来得到
# (Pdb) l
#   4           s3 = s1 + s2 + s1
#   5           s3 = ' ' + s3 + ' '
#   6           return s3
#   7           a = "aaa"

```

```
# 8      pdb.set_trace()
# 9 ->    b = "bbb"
# 10     c = "ccc"
# 11     final = combine(a,b)
# 12     print final
# [EOF]
# (Pdb) b 11
# Breakpoint 1 at /Users/wangmingdong/Desktop/test3.py:11
# (Pdb) c
# > /Users/wangmingdong/Desktop/test3.py(11)<module>()
# -> final = combine(a,b)
# (Pdb) l
# 6          return s3
# 7      a = "aaa"
# 8      pdb.set_trace()
# 9      b = "bbb"
# 10     c = "ccc"
# 11 B->    final = combine(a,b)
# 12     print final

# 《7 进入函数继续调试》
# s---->进入函数里面继续调试，如果使用n表示把一个函数的调用当做一条语句执行
#例如
# (Pdb) l
# 6          return s3
# 7      a = "aaa"
# 8      pdb.set_trace()
# 9      b = "bbb"
# 10     c = "ccc"
# 11 B->    final = combine(a,b)
# 12     print final
# [EOF]
# (Pdb) s
# --Call--
# > /Users/wangmingdong/Desktop/test3.py(3)combine()
# -> def combine(s1,s2):
# (Pdb) l
# 1      import pdb
```

```
# 2
# 3 -> def combine(s1,s2):
# 4         s3 = s1 + s2 + s1
# 5         s3 = ''' + s3 +'''
# 6         return s3
# 7     a = "aaa"
# 8     pdb.set_trace()
# 9     b = "bbb"
# 10    c = "ccc"
# 11 B    final = combine(a,b)
# (Pdb)
```

《8 查看传递到函数中的变量》

a---->调用一个函数时，可以查看传递到这个函数中的所有的参数；a表示arg的意
#例如：

```
# (Pdb) l
# 1      #coding=utf-8
# 2      import pdb
# 3
# 4 -> def combine(s1,s2):
# 5         s3 = s1 + s2 + s1
# 6         s3 = ''' + s3 +'''
# 7         return s3
# 8
# 9     a = "aaa"
# 10    pdb.set_trace()
# 11    b = "bbb"
# (Pdb) a
# s1 = aaa
# s2 = bbb
```

《9 执行到函数的最后一步》

r---->如果在函数中不想一步步的调试了，只是想到这个函数的最后一条语句那个

demo 4

```
In [1]: def pdb_test(arg):
...:     for i in range(arg):
...:         print(i)
...:     return arg
...:
```

In [2]: #在python交互模式中，如果想要调试这个函数，那么可以

In [3]: #采用，pdb.run的方式，如下：

In [4]: import pdb

```
In [5]: pdb.run("pdb_test(10)")
> <string>(1)<module>()
(Pdb) s
--Call--
> <ipython-input-1-ef4d08b8cc81>(1)pdb_test()
-> def pdb_test(arg):
(Pdb) l
  1  ->    def pdb_test(arg):
  2          for i in range(arg):
  3              print(i)
  4          return arg
[EOF]
(Pdb) n
> <ipython-input-1-ef4d08b8cc81>(2)pdb_test()
-> for i in range(arg):
(Pdb) l
  1      def pdb_test(arg):
  2  ->        for i in range(arg):
  3              print(i)
  4          return arg
[EOF]
(Pdb) n
> <ipython-input-1-ef4d08b8cc81>(3)pdb_test()
-> print(i)
(Pdb)
```

```
0
> <ipython-input-1-ef4d08b8cc81>(2)pdb_test()
-> for i in range(arg):
(Pdb)
> <ipython-input-1-ef4d08b8cc81>(3)pdb_test()
-> print(i)
(Pdb)
1
> <ipython-input-1-ef4d08b8cc81>(2)pdb_test()
-> for i in range(arg):
(Pdb)
```

demo 5 运行过程中使用pdb修改变量的值

```
In [7]: pdb.run("pdb_test(1)")
> <string>(1)<module>()
(Pdb) s
--Call--
> <ipython-input-1-ef4d08b8cc81>(1)pdb_test()
-> def pdb_test(arg):
(Pdb) a
arg = 1
(Pdb) l
 1 ->     def pdb_test(arg):
 2             for i in range(arg):
 3                 print(i)
 4             return arg
[EOF]
(Pdb) !arg = 100 #!!!这里是修改变量的方法
(Pdb) n
> <ipython-input-1-ef4d08b8cc81>(2)pdb_test()
-> for i in range(arg):
(Pdb) l
 1     def pdb_test(arg):
 2 ->         for i in range(arg):
 3             print(i)
 4         return arg
```

```
[EOF]
(Pdb) p arg
100
(Pdb)
```

练一练:请使用所学的pdb调试技巧对其进行调试出bug

```
#coding=utf-8
import pdb

def add3Nums(a1,a2,a3):
    result = a1+a2+a3
    return result

def get3NumsAvarage(s1,s2):
    s3 = s1 + s2 + s1
    result = 0
    result = add3Nums(s1,s2,s3)/3

if __name__ == '__main__':
    a = 11
    # pdb.set_trace()
    b = 12
    final = get3NumsAvarage(a,b)
    print final
```

pdb 调试有个明显的缺陷就是对于多线程，远程调试等支持得不够好，同时没有较为直观的界面显示，不太适合大型的 python 项目。而在较大的 python 项目中，这些调试需求比较常见，因此需要使用更为高级的调试工具。

编码风格

错误认知

- 这很浪费时间
- 我是个艺术家
- 所有人都能穿的鞋不会合任何人的脚
- 我擅长制定编码规范

正确认知

- 促进团队合作
- 减少bug处理
- 提高可读性，降低维护成本
- 有助于代码审查
- 养成习惯，有助于程序员自身的成长

pep8 编码规范

Python Enhancement Proposals : python改进方案

<https://www.python.org/dev/peps/>

python基础语言

The screenshot shows the Python Developer's Guide with a navigation bar at the top. Below it is a sidebar for 'The PSF' (Python Software Foundation) and the main content area for PEP 0.

PEP 0 -- Index of Python Enhancement Proposals (PEPs)

PEP:	0
Title:	Index of Python Enhancement Proposals (PEPs)
Last-Modified:	2016-10-06
Author:	David Goodger <goodger at python.org>, Barry Warsaw <barry at python.org>
Status:	Active
Type:	Informational
Created:	13-Jul-2000

Introduction

This PEP contains the index of all Python Enhancement Proposals, known as PEPs. PEP numbers are assigned by the PEP editors, and once assigned are never changed[1]. The Mercurial history[2] of the PEP texts represent their historical record.

pep8 官网规范地址

<https://www.python.org/dev/peps/pep-0008/>

The screenshot shows the Python Style Guide (PEP 8) page. It includes a sidebar with links like 'ABOUT', 'NEWS', 'DOCUMENTATION', 'DOWNLOAD', 'COMMUNITY', 'FOUNDATION', and 'CORE DEVELOPMENT'. The main content area displays the PEP 8 details and its table of contents.

PEP: 8
Title: Style Guide for Python Code
Version: c451868df657
Last-Modified: 2016-06-08 10:43:53 -0400 (Wed, 08 Jun 2016)
Author: Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status: Active
Type: Process
Content-Type: text/x-rst
Created: 05-Jul-2001
Post-History: 05-Jul-2001, 01-Aug-2013

Contents

- Introduction
- A Foolish Consistency is the Hobgoblin of Little Minds
- Code lay-out
 - Indentation
 - Tabs or Spaces?
 - Maximum Line Length
 - Should a line break before or after a binary operator?
 - Blank Lines

Guido的关键点之一是：代码更多是用来读而不是写。编码规范旨在改善Python代码的可读性。

风格指南强调一致性。项目、模块或函数保持一致都很重要。

每级缩进用4个空格。

括号中使用垂直隐式缩进或使用悬挂缩进。后者应该注意第一行要没有参数，后续行要有缩进。

- Yes

```
# 对准左括号
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# 不对准左括号，但加多一层缩进，以和后面内容区别。
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)

# 悬挂缩进必须加多一层缩进。
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

- No

```
# 不使用垂直对齐时，第一行不能有参数。
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# 参数的缩进和后续内容缩进不能区别。
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

4个空格的规则是持续行可选的。

```
# 悬挂缩进不一定是4个空格
foo = long_function_name(
```

```

var_one, var_two,
var_three, var_four)

if语句跨行时，两个字符关键字(比如if)加上一个空格，再加上左括号构成了很好的缩进。
# 没有额外缩进，不是很好看，个人不推荐。
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# 添加注释
if (this_is_one_thing and
    that_is_another_thing):
    # Since both conditions are true, we can frobnicate.
    do_something()

# 额外添加缩进，推荐。
# Add some extra indentation on the conditional continuation line.
if (this_is_one_thing
    and that_is_another_thing):
    do_something()

```

右边括号也可以另起一行。有两种格式，建议第2种。

```

# 右括号不回退，个人不推荐
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)

# 右括号回退
my_list = [
    1, 2, 3,

```

```
    4, 5, 6,  
]  
result = some_function_that_takes_arguments(  
    'a', 'b', 'c',  
    'd', 'e', 'f',  
)
```

空格或Tab?

- 空格是首选的缩进方法。
- Tab仅仅在已经使用tab缩进的代码中为了保持一致性而使用。
- Python 3中不允许混合使用Tab和空格缩进。
- Python 2的包含空格与Tab和空格缩进的应该全部转为空格缩进。

最大行宽

- 限制所有行的最大行宽为79字符。
- 文本长块，比如文档字符串或注释，行长度应限制为72个字符。

空行

- 两行空行分割顶层函数和类的定义。
- 类的方法定义用单个空行分割。
- 额外的空行可以在必要的时候用于分割不同的函数组，但是要尽量节约使用。
- 额外的空行可以在必要的时候在函数中用于分割不同的逻辑块，但是要尽量节约使用。

源文件编码

- 在核心Python发布的代码应该总是使用UTF-8(ASCII在Python 2)。
- Python 3(默认UTF-8)不应有编码声明。

导入在单独行

- Yes:

```
import os
import sys
from subprocess import Popen, PIPE
```

- No:

```
import sys, os
```

- 导入始终在文件的顶部，在模块注释和文档字符串之后，在模块全局变量和常量之前。
- 导入顺序如下：标准库进口,相关的第三方库，本地库。各组的导入之间要有空行。

禁止使用通配符导入。

通配符导入(`from import *`)应该避免，因为它不清楚命名空间有哪些名称存，混淆读者和许多自动化的工具。

字符串引用

- Python中单引号字符串和双引号字符串都是相同的。注意尽量避免在字符串中的反斜杠以提高可读性。
- 根据PEP 257, 三个引号都使用双引号。

括号里边避免空格

```
# 括号里边避免空格
# Yes
spam(ham[1], {eggs: 2})
# No
spam( ham[ 1 ], { eggs: 2 } )
```

逗号，冒号，分号之前避免空格

```
# 逗号, 冒号, 分号之前避免空格  
# Yes  
if x == 4: print x, y; x, y = y, x  
# No  
if x == 4 : print x , y ; x , y = y , x
```

索引操作中的冒号当作操作符处理前后要有同样的空格(一个空格或者没有空格, 个人建议是没有。)

```
# Yes  
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]  
ham[lower:upper], ham[lower:upper:], ham[lower::step]  
ham[lower+offset : upper+offset]  
ham[: upper_fn(x) : step_fn(x)], ham[::- step_fn(x)]  
ham[lower + offset : upper + offset]  
# No  
ham[lower + offset:upper + offset]  
ham[1: 9], ham[1 :9], ham[1:9 :3]  
ham[lower : : upper]  
ham[ : upper]
```

函数调用的左括号之前不能有空格

```
# Yes  
spam(1)  
dct['key'] = lst[index]  
  
# No  
spam (1)  
dct ['key'] = lst [index]
```

赋值等操作符前后不能因为对齐而添加多个空格

```
# Yes
```

```
x = 1
y = 2
long_variable = 3

# No
x      = 1
y      = 2
long_variable = 3
```

二元运算符两边放置一个空格

涉及 =、符合操作符 (+= , -= 等)、比较(== , < , > , != , <> , <= , >=)、in , not in , is , is not)、布尔(and , or , not)。

优先级高的运算符或操作符的前后不建议有空格。

```
# Yes
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)

# No
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

关键字参数和默认值参数的前后不要加空格

```
# Yes
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

```
# No
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

通常不推荐复合语句(Compound statements: 多条语句写在同一行)。

```
# Yes
if foo == 'blah':
    do_blahting()
do_one()
do_two()
do_three()

# No
if foo == 'blah': do_blahting()
do_one(); do_two(); do_three()
```

尽管有时可以在if/for/while 的同一行跟一小段代码，但绝不要跟多个子句，并尽量避免换行。

```
# No
if foo == 'blah': do_blahting()
for x in lst: total += x
while t < 10: t = delay()

更不是:

# No
if foo == 'blah': do_blahting()
else: do_non_blahting()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                           list, like, this)
```

```
if foo == 'blah': one(); two(); three()
```

避免采用的名字

决不要用字符'I'(小写字母el), 'O'(大写字母oh), 或 'l'(大写字母eye) 作为单个字符的变量名。一些字体中，这些字符不能与数字1和0区别。用'L'代替'I'时。

包和模块名

模块名要简短，全部用小写字母，可使用下划线以提高可读性。包名和模块名类似，但不推荐使用下划线。