<http://www.pythonforbeginners.com/code-snippets-source-code/python-code-examples>

<http://www.pythontab.com>

<http://python.usyiyi.cn>

基础文档教材

<http://python.usyiyi.cn/translate/python_352/tutorial/datastructures.html>

分拆参数列表

相反的情况发生在参数已经在列表或元组中，但是需要为需要单独的位置参数的函数调用解包时。例如，内建的[range()](http://python.usyiyi.cn/documents/python_352/library/stdtypes.html#range)函数期待单独的start和stop参数。如果它们不能单独地获得，可以编写带有\*操作的函数调用，来从一个列表或元组分拆出参数：

>>>

>>> list(range(3, 6)) # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(\*args)) # call with arguments unpacked from a list
[3, 4, 5]

同样的风格，字典可以通过\*\*操作传递关键字参数：

>>>

>>> def parrot(voltage, state='a stiff', action='voom'):
... print("-- This parrot wouldn't", action, end=' ')
... print("if you put", voltage, "volts through it.", end=' ')
... print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(\*\*d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !

------------------------------------------------------------

def example():

""" 函数的一些文档使用说明.

可以换行，多行.(3个单引号或双引号)

"""

函数的文档可以使用example.\_\_doc\_\_ 获取。

----------------------------------------------------------------

函数注解：就是给普通函数加上一个注解或默认值

>>> **def** func(a:'spam'=4,b:(1,10)=5,c:float=6)->int:

    **return** a+b+c

'spam'是参数a的注释，=4是给a的默认值，(1,10)是了的注释,=5是b的默认值，c:float=6 指c是一个float并且默认值是6, ->int代表返回类型.

对于参数的注解出现在紧随参数名之后的冒号之后;对于返回值,它们编写于紧跟在参数列表之后的一个 ->  之后.

可以通过 func.\_\_annotations\_\_ 查看。

|  |
| --- |
| >>> vec = [-4, -2, 0, 2, 4]  >>> # create a new list with the values doubled  >>> [x\*2 for x in vec]  [-8, -4, 0, 4, 8]  >>> # filter the list to exclude negative numbers  >>> [x for x in vec if x >= 0]  [0, 2, 4]  >>> # apply a function to all the elements  >>> [abs(x) for x in vec]  [4, 2, 0, 2, 4]  >>> # call a method on each element  >>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']  >>> [weapon.strip() for weapon in freshfruit]  ['banana', 'loganberry', 'passion fruit']  >>> # create a list of 2-tuples like (number, square)  >>> [(x, x\*\*2) for x in range(6)]  [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]  >>> # the tuple must be parenthesized, otherwise an error is raised  >>> [x, x\*\*2 for x in range(6)]  File "<stdin>", line 1, in ?  [x, x\*\*2 for x in range(6)]  ^  SyntaxError: invalid syntax  >>> # flatten a list using a listcomp with two 'for'  >>> vec = [[1,2,3], [4,5,6], [7,8,9]]  >>> [num for elem in vec for num in elem]  [1, 2, 3, 4, 5, 6, 7, 8, 9] |

嵌套的列表推导式

列表推导式的第一个表达式可以是任何表达式，包括另外一个列表推导式。

考虑下面的3×4矩阵的示例，其被实现为长度为4的3个列表的列表：

>>>

**>>>** matrix = [
**...**  [1, 2, 3, 4],
**...**  [5, 6, 7, 8],
**...**  [9, 10, 11, 12],
**...** ]

下面的列表推导式将转置行和列：

>>>

**>>>** [[row[i] **for** row **in** matrix] **for** i **in** range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

正如在前一节中我们所见，嵌套的列表推导式在它后面的[for](http://python.usyiyi.cn/documents/python_352/reference/compound_stmts.html#for)上下文中求值，所以这个例子等效于：

>>>

**>>>** transposed = []
**>>> for** i **in** range(4):
**...**  transposed.append([row[i] **for** row **in** matrix])
**...**
**>>>** transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

展开后，和下面的代码相同：

>>>

**>>>** transposed = []
**>>> for** i **in** range(4):
**...**  *# the following 3 lines implement the nested listcomp*
**...**  transposed\_row = []
**...**  **for** row **in** matrix:
**...**  transposed\_row.append(row[i])
**...**  transposed.append(transposed\_row)
**...**
**>>>** transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

在现实世界中，您应该喜欢内建函数复杂的流语句。[zip()](http://python.usyiyi.cn/documents/python_352/library/functions.html#zip)函数对这个使用场景做得非常好：

>>>

**>>>** list(zip(\*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]

-----------------------------------------------------

元组由一组用逗号分隔的值组成。元组是不可变的。

t = 12345, 54321, 'hello!' //取个数t[0]

也可以反向操作 x,y,z = t //从 t中的3个元组分别赋值给x,y,z

元组嵌套：

>>>u = t, (1,3,5,6)

>>>u

>>> ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

----------------------------------------------------------

集合:

**>>>** basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}

**>>>** 'orange' **in** basket *# fast membership testing*
True

**>>>** *# Demonstrate set operations on unique letters from two words*
**...**
**>>>** a = set('abracadabra')
**>>>** b = set('alacazam')
**>>>** a *# unique letters in a*
{'a', 'r', 'b', 'c', 'd'}
**>>>** a - b *# letters in a but not in b*
{'r', 'd', 'b'}
**>>>** a | b *# letters in either a or b*
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
**>>>** a & b *# letters in both a and b*
{'a', 'c'}
**>>>** a ^ b *# letters in a or b but not both*
{'r', 'd', 'b', 'm', 'z', 'l'}

类似于[列表推导式](http://python.usyiyi.cn/documents/python_352/tutorial/datastructures.html#tut-listcomps)，集合也支持推导式：

>>>

**>>>** a = {x **for** x **in** 'abracadabra' **if** x **not** **in** 'abc'}
**>>>** a
{'r', 'd'}

----------------------------------------------

字典

在字典上执行list(d.keys())将返回字典中用到所有的键的一个列表，但是没有顺序（如果想要排序，只需使用sorted(d.keys())）。[[2]](http://python.usyiyi.cn/documents/python_352/tutorial/datastructures.html#id4)若要检查一个键是否在字典中，可以使用[in](http://python.usyiyi.cn/documents/python_352/reference/expressions.html#in)关键字。

将元组列表转成字典：

**>>>** dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}

字典推导式可以用于从任意键和值表达式创建字典：

**>>>** {x: x\*\*2 **for** x **in** (2, 4, 6)}
{2: 4, 4: 16, 6: 36}

当键是简单字符串时，使用关键字参数指定对有时更容易：

**>>>** dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}

--------------------------------------------------------

循环的技巧

当循环遍历字典时，键和对应的值可以使用items()方法同时提取出来。

**>>>** knights = {'gallahad': 'the pure', 'robin': 'the brave'}
**>>> for** k, v **in** knights.items():
**...**  print(k, v)
**...**
gallahad the pure
robin the brave

当遍历一个序列时，使用[enumerate()](http://python.usyiyi.cn/documents/python_352/library/functions.html#enumerate)函数可以同时得到位置索引和对应的值。

**>>> for** i, v **in** enumerate(['tic', 'tac', 'toe']):
**...**  print(i, v)
**...**
0 tic
1 tac
2 toe

同时遍历两个或更多的序列，使用[zip()](http://python.usyiyi.cn/documents/python_352/library/functions.html#zip)函数可以成对读取元素。

**>>>** questions = ['name', 'quest', 'favorite color']
**>>>** answers = ['lancelot', 'the holy grail', 'blue']
**>>> for** q, a **in** zip(questions, answers):
**...**  print('What is your *{0}*? It is *{1}*.'.format(q, a))
**...**
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.

增加路径：

**>>> import** **sys**
**>>>** sys.path.append('/ufs/guido/lib/python')

内置函数 [dir()](http://python.usyiyi.cn/documents/python_352/library/functions.html#dir) 用来找出模块中定义了哪些名字。它返回字符串的排序列表：

>>>import sys

>>>dir(sys)

如果不带参数， [dir()](http://python.usyiyi.cn/documents/python_352/library/functions.html#dir) 列出当前已定义的名称：

包内引用

如果一个包是子包（比如例子中的 sound 包），你可以使用绝对导入来引用兄弟包的子模块。例如，如果模块sound.filters.vocoder需要使用sound.effects包中的echo模块，它可以使用from sound.effects import echo。

你还可以使用from module import name形式的导入语句写成相对导入。这些导入使用前导点来指示相关导入中涉及的当前和父包。以 surround 模块为例，你可以使用：

**from** **.** **import** echo
**from** **..** **import** formats
**from** **..filters** **import** equalizer

请注意，相对导入基于当前模块的名称。因为主模块的名字总是 "\_\_main\_\_" ，Python 应用程序的主模块应该总是用绝对导入。

类和实例变量

一般来说，实例变量是每个实例唯一的数据，类变量是由类的所有实例共享的属性和方法：

**class** **Dog**:
kind = 'canine' *# class variable shared by all instances*
**def** \_\_init\_\_(self, name):
self.name = name *# instance variable unique to each instance*
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind *# shared by all dogs*
'canine'
>>> e.kind *# shared by all dogs*
'canine'
>>> d.name *# unique to d*
'Fido'
>>> e.name *# unique to e*
'Buddy'

正如在 [名称和对象](http://python.usyiyi.cn/documents/python_352/tutorial/classes.html#tut-object) 讨论的， [可变](http://python.usyiyi.cn/documents/python_352/glossary.html#term-mutable) 对象，例如列表和字典，的共享数据可能带来意外的效果。例如，以下代码中的**trick**列表不应用作类变量，因为所有**Dog**实例都将共享单个列表：

**class** **Dog**:
tricks = [] *# mistaken use of a class variable*
**def** \_\_init\_\_(self, name):
self.name = name
**def** add\_trick(self, trick):
self.tricks.append(trick)
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add\_trick('roll over')
>>> e.add\_trick('play dead')
>>> d.tricks *# unexpectedly shared by all dogs*
['roll over', 'play dead']

正确的类设计应该使用实例变量：

**class** **Dog**:
**def** \_\_init\_\_(self, name):
self.name = name
self.tricks = [] *# creates a new empty list for each dog*
**def** add\_trick(self, trick):
self.tricks.append(trick)
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add\_trick('roll over')
>>> e.add\_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']

数据属性会覆盖同名的方法属性；为了避免意外的命名冲突，这在大型程序中可能带来极难发现的 bug，使用一些约定来减少冲突的机会是明智的。可能的约定包括大写方法名称，使用小的唯一字符串（可能只是下划线）为数据属性名称添加前缀，或者对数据属性使用方法和名词的动词。

通常，方法的第一个参数称为 self 。这仅仅是一个约定：名字 self 对 Python 而言绝对没有任何特殊含义。但请注意，不遵循约定，你的代码对其他Python程序员的可读性可能较差，也可以设想一个依赖于这样的约定的**类浏览器**程序。

作为类属性的任何函数对象定义该类的实例的方法。函数定义代码不一定非得定义在类中：也可以将一个函数对象赋值给类中的一个局部变量。例如：

*# Function defined outside the class*
**def** f1(self, x, y):
**return** min(x, x+y)
**class** **C**:
f = f1
**def** g(self):
**return** 'hello world'
h = g

现在 f、g 和 h 都是类 C中引用函数对象的属性，因此它们都是 C 的实例的方法 —— h 完全等同于 g 。请注意，这种做法通常只会使阅读程序的人产生困惑。

方法可以通过使用 self 参数的方法属性，调用其他方法：

**class** **Bag**:
**def** \_\_init\_\_(self):
self.data = []
**def** add(self, x):
self.data.append(x)
**def** addtwice(self, x):
self.add(x)
self.add(x)

**class** **DerivedClassName**(modname.BaseClassName):

派生类定义的执行与基类的执行相同。当构建类对象时，将记住基类。这用于解析属性引用：如果在类中找不到请求的属性，则搜索继续查找基类。如果基类本身是从某个其他类派生的，则此规则将递归应用。

派生类可以重写基类中的方法。因为方法在调用同一对象的其他方法时没有特殊的权限，所以调用在同一个基类中定义的另一个方法的基类的方法可能最终会调用覆盖它的派生类的方法。

派生类中的重写方法实际上可能希望扩展而不是简单地替换同名的基类方法。有一个简单的方法可以直接调用基类方法：只要调用BaseClassName.methodname(self, arguments) 。这有时对客户有用。（要注意只有 BaseClassName 在同一全局作用域定义或导入时才能这样用。）

isinstance()检查实例类型。issubclass()来检查继承。

多继承 class DerivedClassname(Base1, Base2, Base3):

对于大多数情况，在最简单的情况下，您可以将从父类继承的属性的搜索视为深度优先，从左到右，不在同一类中在层次结构中存在重叠的情况下进行两次搜索。因此，如果在 DerivedClassName 中找不到属性，它搜索 Base1 ，然后（递归）基类中的 Base1 ，如果没有找到，它会搜索 Base2 ，依此类推。查找方法从继承的基类左到右。

私有变量：

Python中并不存在所谓只能在对象内部才能访问的“私有”实例变量。然而，有一项大多数 Python 代码都遵循的习惯：带有下划线（例如\_spam ）前缀的名称应被视为非公开的 API 的一部分（无论是函数、 方法还是数据成员）。

由于存在一种合理的类私有成员使用场景（例如为了避免名称与子类所定义名称的冲突），Python 对这种机制提供了简单的支持，即所谓“名称重整” （**name mangling**）。\_\_spam 形式的任何标识符（前面至少两个下划线，后面至多一个下划线）将被替换为\_classname\_\_spam， classname 是当前类的名字。这种替换是在不考虑标识符句法位置的情况下完成的，只要它出现在类的定义内即可。

名称整理有助于让子类重写方法，而不会打断类内方法调用。例如：

**class** **Mapping**:
**def** \_\_init\_\_(self, iterable):
self.items\_list = []
self.\_\_update(iterable)
**def** update(self, iterable):
**for** item **in** iterable:
self.items\_list.append(item)
\_\_update = update *# private copy of original update() method*
**class** **MappingSubclass**(Mapping):
**def** update(self, keys, values):
*# provides new signature for update()*
*# but does not break \_\_init\_\_()*
**for** item **in** zip(keys, values):
self.items\_list.append(item)

请注意名称改编的目的主要是避免发生意外；访问或者修改私有变量仍然是可能的。

有时，使用类似于Pascal“record”或C“struct”的数据类型，捆绑几个命名的数据项是有用的。一个空类定义会很好：

**class** **Employee**:
**pass**
john = Employee() *# Create an empty employee record*
*# Fill the fields of the record*
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000

一段需要特定抽象数据类型的Python代码，通常可以传递一个类来模拟该数据类型的方法。例如，如果你有一个用于从文件对象中格式化数据的函数，你可以定义一个带有 read() 和 readline() 方法的类，以此从字符串缓冲读取数据，然后将该类的对象作为参数传入前述的函数。

实例的方法对象也有属性： m.\_\_self\_\_是具有方法 m() 的实例对象， m.\_\_func\_\_ 是方法的函数对象。

闭包: 非本地变量 nonlocal, 全局变量 global

g = lambda x:2 \* x 冒号前面的x代表匿名函数的参数；lambda就类似一个匿名函数.调用g(5);

g = lambda x,y : x+y 两个参数的匿名函数。

filter(Function or None, iterable)

map(Function, iterable) 加工处理

斐波那契：除去1，2，从3开始每个数等于前2个数的和。汉诺塔用递归。

dict.fromkeys 创建字典；

dict.fromkeys((1,2,3)) 得到 {1:None,2:None,3:None}

doct.fromkeys((1,2,3),(Number)) 得到 {1:Number, 2:Number, 3:Number}

可以用 in or not in 判断存不存在。

dir(dict)查看内置属性；id(dir)查看内存地址; a.update(b) 把a字典加入到b字典

dict(((a,b),(b,c)))把元组转成字典

frozeSet有序的set; pickle模块具有持久性，类似java的序列化。pickle.dump(),文件 = open(xx,'wb')；加载pickle.load(文件);

异常:except(OSError, XXError); raise产生异常。

while true:

pass

else: //跳出循环后执行

pass

else可加try except中和for中。

简洁的with可以自动帮你关闭文件，如果你有打开文件可以使用

with open('xx.txt', 'w') as f:

f.read();

可视化模块:EasyGui,

class Xxx:类，实例 x = Xxx()

继承在类中加入父类 class A(B) A继承B,方法中要加入(self),self相对于this的指针,类是图纸，实例才是实物，self自身做为第一个参数。

\_\_init\_\_(self)类似构造函数

私有：在名字前加上\_\_双下划线

继承-》构造函数重写-》调用未绑定的方法或调用super方法。

\_\_init\_\_(self):

父类.\_\_init\_\_(self)

self.xx

推荐使用super().\_\_init\_\_()

类属性值的改变会影响实例中它的值。如果属性名和方法名一样，属性名会覆盖方法。

类中如果有以下定义x=property(get,set,del) 那么可以直接用实例进行访问 get 可以用 实例.x ，set就直接赋值 实例.x= ,del 实例.x

魔法方法总是被\_\_双下划线所包围; 实例化时总是先运行\_\_new\_\_;如果继承不可以改变对象，可以使用\_\_new\_\_进行重写；

class CapStr(str):

def \_\_new\_\_(cls, string):

string = string.upper()

return str.\_\_new\_\_(cls, string)

垃圾回收时调用 \_\_del\_\_ 内置方法。

\_\_str\_\_类似java toString. \_\_str\_\_

反运算，右操作数是左操作数的子类，并且该子类提供了操作的反射方法。则该方法将左操作数的非反射方法前被调用。

取属性先调用\_\_getAttribute(..),找不到再调用\_\_getAttr取一个不存在的属性。

属性赋值操作执行\_\_setAttr.

yield生成器-协同程序就是可以运行的独立函数调用，函数可以暂停或者挂起，并在需要的时候从程序离开的地方继续或者重新开始。

e = (i for i in range(10))得到一个生成器。

容器->数据的封装； 函数->语句的封装； 类->方法和属性的封装；

主程序中\_\_name\_\_等于\_\_main\_\_,如果是被导入到其它模块做为模块，它的模块名为\_\_name\_\_。

import sys --> sys.path环境变量

最佳模块存放位置\lib\site-packages

将自己项目模块路径加到path ,sys.path.append('路径');

文件夹就是包名，在文件夹中创建一个\_\_init\_\_.py的模块文件，内容为空。

模块放在一起，\_\_init\_\_.py的作用是告诉python把这个文件夹当成包管理。 导入 import 包名.模块名

\_\_all\_\_显示此模块可对外的函数，模块中对外的属性用\_\_all\_\_设置。可以在\_\_init\_\_.py里写入内容\_\_all\_\_ = ['文件名'] ,可以把包里的模块放开出去，让其他人可以导入。还可以在这个文件里写其他代码。**导入模块时会运行\_\_init\_\_.py里的内容。有\_\_init\_\_.py文件才是包**。

--------------------------------------------

\_\_all\_\_ = ['sendmsg']

from . import sendmsg #在当前包中导入sendmsg,这样其他模块导入这个模块的时候

#可以直接使用xx.sendmsg

---------------------------------------------

urllib.request用来api访问。

import json

json.loads(字典) 当字典转成json;

爬虫框架 scrapy, Gui界面 :Tkinter, 游戏开发用pygame

\_\_all\_\_变量 ，模块中可能会有一大堆其他程序不需要或不想要的变量，函数或类，\_\_all\_\_会将它们过滤出去。如果没有设定\_\_all\_\_，用import \*语句默认会将导入模块中所有不以下划线开头的全局名称。

\_\_all\_\_ = ['copy','test'] 设置哪里方法，变量可以被导入。

查找源码 模块.\_\_file\_\_

python文件转成exe文件:

使用PyInstaller <http://www.pyinstaller.org/>

PyInstaller Quickstart

Install PyInstaller from PyPI:

pip install pyinstaller

Go to your program’s directory and run:

pyinstaller yourprogram.py

**如果想运行的时候不弹出console，可以加参数 --noconsole**

This will generate the bundle in a subdirectory called dist.

**Python提供了可变参数\*args和关键字参数\*\*kwargs**

**def test\_var\_args(f\_arg, \*argv):**

**print "first normal arg:", f\_arg**

**for arg in argv:**

**print "another arg through \*argv :", arg**

def greet\_me(\*\*kwargs):

if kwargs is not None:

for key, value in kwargs.iteritems():

print "%s == %s" %(key,value)

>>> greet\_me(name="yasoob")

name == yasoob

<https://pythontips.com/2013/08/04/args-and-kwargs-in-python-explained/>

装饰器本质上是一个Python函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象。它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码并继续重用。

装饰器的作用就是**为已经存在的函数或对象添加额外的功能**。

def debug(func):
def wrapper(something): # 指定一毛一样的参数
print "[DEBUG]: enter {}()".format(func.\_\_name\_\_)
return func(something)
return wrapper # 返回包装过函数
@debug
def say(something):
print "hello {}!".format(something)

带参数的被装饰者

def debug(func):
def wrapper(\*args, \*\*kwargs): # 指定宇宙无敌参数
print "[DEBUG]: enter {}()".format(func.\_\_name\_\_)
print 'Prepare and say...',
return func(\*args, \*\*kwargs)
return wrapper # 返回
@debug
def say(something):
print "hello {}!".format(something)

**基于类实现的装饰器**

装饰器函数其实是这样一个接口约束，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。只要某个对象重载了\_\_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

像\_\_call\_\_这样前后都带下划线的方法在Python中被称为内置方法，有时候也被称为魔法方法。重载这些魔法方法一般会改变对象的内部行为。上面这个例子就让一个类对象拥有了被调用的行为。

回到装饰器上的概念上来，装饰器要求接受一个callable对象，并返回一个callable对象（不太严谨，详见后文）。那么用类来实现也是也可以的。我们可以让类的构造函数\_\_init\_\_()接受一个函数，然后重载\_\_call\_\_()并返回一个函数，也可以达到装饰器函数的效果。

class logging(object):
def \_\_init\_\_(self, func):
self.func = func
def \_\_call\_\_(self, \*args, \*\*kwargs):
print "[DEBUG]: enter function {func}()".format(
func=self.func.\_\_name\_\_)
return self.func(\*args, \*\*kwargs)
@logging
def say(something):
print "say {}!".format(something)

<https://www.cnblogs.com/cicaday/p/python-decorator.html>

[执行命令](https://www.cnblogs.com/cicaday/p/python-decorator.html)

os.system(cmd) 会阻塞后面的操作，返回退出码。或os.popen(cmd)返回值是脚本执行过程中的输出内容。os.popen（）可以实现一个“管道”，从这个命令获取的值可以继续被调用

字节表达式 bytes = b'this is byte'

>python2 xx.py //用python2的版本运行

>python3 xx.py //用python3运行

.ipython 就可以在linux使用linux的指令和python命令一起共存，或 ipython3

如果python中有中文，为了防止乱码需要指写编码，需要在首行加入以下代码:

#-\*- coding:utf-8 -\*- 或者 #coding=utf-8 推荐使用第一种

如果变量第一个出现则是定义了一个变量，不是第一次则表示得新赋值。

运算符 // 得到商，%得到余数, \*\* 幂 2\*\*3 = 2的3次方

1M=1024K ,1K=1024Byte(字节),1个字节存的最大值是255

双引号里的字符串每个字符占一个字节。

底层占位其实多一位 "/0"代表结束。

组成新字符串的2种方式：

f=“=="+"a"+"===" ,f = "===%s==="%(a变量)

字符串切片[起始位置:结束位置:步长] a="abcdefg" a[0:-2:2] 从a开始到e结束包括e,隔2个字符才取，-2表示从尾算起。

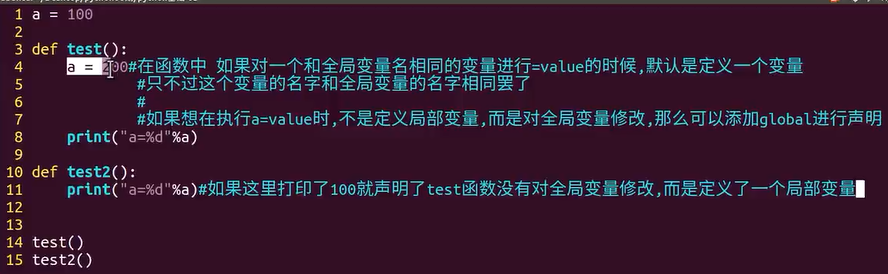
如果想字符串倒序也可以用切片实现 a[-1::-1] [::-1]

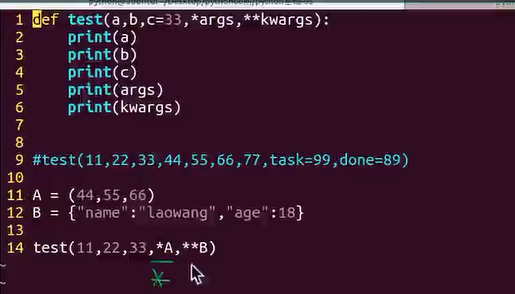
拆包:

c,d = a = (11,12) 元组赋值，11给c, 12给d,a就是一个元组

for A, B in 元组：

元组是不可以修改的，它只是只读的，元组是括号中的元素用逗号隔开如(1,2,3,4,5,6)





如果调用test(11,22,33,A,B)那么A,B变量的值会被当成args参数。

所以传递时加上\*A是对A变量进行拆包，把值给args; \*\*B是对B变量进行拆包，将B变量的字典 数据传给kwargs.

私有方法:以双下划线\_\_开头的方法为私有的，外面不能访问。

父类调用可以使用:

1. 父类.方法(self)

2. super().方法()

继承：如果调用的是继承的父类中的公有方法，可以在这个父类的公有方法中访问父类中的私有属性和私有方法；但是如果在子类中实现了一个公有方法，那么这个方法是不能够调用继承的父类中的私有方法和私有属性。

如果一个变量已经在全局变更的位置定义了，此时还想在函数中对这个全局变量进行修改的话。那么，仅仅是修改变量值还不够，此时变量是一个局部变量，跟全局变量名称相同而已。如：

A= 0 #全局

def getA():

A = 100 #是局部变量，如果你在局部变量中修改全局变量，要在函数中A = 100前加上[ global A ]这样表明要修改全局变量。

---------

A = 0

def getA():

global A

A = 100 #此时全局变量就变成了100

def printA():

print(A) #取全局变量的A

程序是从上往下执行，全局变量最好放在调用方法的前面。全局变量与局部变量名字相同，只要局部函数没有使用global，则函数调用局部变量。

**建议全局变量定义为g\_变量名,避免与局部变量名产生冲突。**

在函数中如果没有对全局变量进行修改时可以不加global,要修改一定要加上，不过不加也不为过，加了更容易读懂。

**全局变量的字典,列表，可以在函数中直接操作，不需要加global.**

**函数可以返回多个值:**

def x():

return 1,2 #其实是返回一个元组

可以将返回值赋值给多个变量如 a,b = x() #将x函数的返回的元组的第一个值给a,第二个值给b.

缺省参数：

def x(a, b='abc') #缺省参数不传就是'abc',没有缺省值的参数要放在前面。

def test(a, b=22, c=33)

test(11, c=10) #没有传参数b,那么b就为缺省参数值为22.c=10为命名参数

def test1(a, b, \*args) #args代表不定长参数，不确定参数个数。

**('A')不是元组,('A',)才是元组。元组一定至少有一个逗号(,)**

def test(a, b, c=1, \*args, \*\*kwards) #\*\*kwards为字典参数

调用传参时，用命名参数的都会给到kwards参数中去

test(11,12, 13,14,task=99,done=98) #11给a,12给b,13给c, 14给\*args, task和done给到了字典参数kwards

Python引用，python里所有赋值等号通通都是引用。

**可变类型：定义完后随便修改（列表，字典）**

**不可变类型：定义完后不能修改。（数字，字符串，元组）**

字典中数字，字符串，元组都可以当键，不可变类型当键，因为它们的hash码不会变

对存字典的列表排序，列表.sort(key=lambda x:x['key值名'])按key或其它排序。

Python交换2个变量可以用 a,b = b,a

a+=a 不等于 a=a+a 临时变量 a =a + a 等于左边的a是临时局部变量，先算等号右边，然后赋值给a

**读文件最好不用read,因为文件可能内容很大，最好一点点的读read(1024); os.getcwd()可以查看被操作文件的当前路径。**

类中定义方法必须传递参数self;可以随意在外部给实例增加属性，不过这样不是很好的方式。

class Base(object): 代表是新式类，不写object代表经典类，反正写不写都是继承object.写上的话会有一些新功能。推荐写上。

**类名.\_\_mro\_\_可以查看方法搜索顺序，找到就停止搜索，子类搜索父类的顺序**。

如果想执行指定父类中的某个方法，可以直接用 父类名.方法名(self) 显示指定。或者 super(当前子类名,self).方法名

python即是面向对象又是面向过程

**类属性：定义在class内和方法外，属于类对象，并且多个实例对象之间共享同一个类属性。\_\_init\_\_(self)中定义的属性**

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

self.name = name

实例属性：实例属性和具体的某个实例对象有关系，并且一个实例对象和另外一个实例对象是不共享属性的。

在实例方法中获取类属性使用 类名.属性



静态方法可以没有参数。其实方法中的self可以用其它名代替，只是业界统一用self.

如果给实例变量赋值一个同类属性一样的属性名称，则你则不能通过实例对象获得类属性的值，因为你的实例属性已经存在了同类名称一样的属性。最好不要用实例去访问或操作类属性。

**实例属性可以随意调用函数**

**\_\_new\_\_(cls)创建对象时执行**

**def \_\_new\_\_(cls):#重写\_\_new\_\_方法一定要调用父类的\_\_new\_\_**

**return object.\_\_new\_\_(cls)**

class Dog:

pass

dog = Dog() #要做3件事，1.调用\_\_new\_\_方法来创建对象，然后找了一个变量来接收\_\_new\_\_的返回值，这个返回值表示创建出来的对象的引用。2.\_\_init\_\_刚刚创建出来的对象的引用。3.返回对象的引用

**\_\_new\_\_负责创建；\_\_init\_\_负责初始化;\_\_new\_\_和\_\_init\_\_两个完成构造方法要做的事情。**

python3捕获多个异常: except (OSError, FileNotFoundError) as ret: ret是指向抛出异常那个类实例的引用。可以调用对象的属性。

-----------------------------------------------------------------------------

#如果导入这个模块的方式是 from 模块名 import \* ,那么仅仅会导入\_\_all\_\_的列表中包含的名字

#为了防止导入所有用不到的方法，这个\_\_all\_\_是一个全局变量。可以方法名，类名，全局属性名.模块名就是py的文件名。

\_\_all\_\_ = ["test1","Test"]

def test1():

print("----test-1-----")

def test2():

print("----test-2-----")

num = 100

class Test(object):

pass

----------------------------------------------------------------------------------------------

**发布模块：**

将模块（此处名为nester）写好后，与setup.py放入同一个文件夹中；

//setup.py

copycode.gif

复制代码

from distutils.core import setup
setup(
name="发布名，一般同模块名",
version="版本号",
py\_modules="模块列表",
"""参数,其他参数根据需求添加"""
)

copycode.gif

复制代码

构建发布文件：在文件夹下打开终端（shift+右键，选择之），输入：python setup.py build 然后再 python setup.py sdist

安装发布到本地副本： 在终端下继续，输入：setup.py install

发布安装后会在该文件夹下生成build/lib目录，改目录下保存nester.py ,dist目录下保存发布包；nester.pyc（也有可能是在\_\_pycache\_\_目录下）是编译版本的代码，MANIFEST包含发布中的文件列表；

此时已可以使用模块中的函数，使用时先要

import 模块名

模块名.函数名（参数）//这是由于要对命名空间做出限定，以便编译器判断是那个命名空间中的这个函数（会有同名函数位于不同命名空间中）；

也可以

from 模块名 import 函数名//使用此方法是将指定函数导入、增加到当前命名空间中来，如此，不必使用命名空间限定

函数名（参数）

 //此方法的弊端是，若该命名空间中原有同名函数，会被后导入的覆盖

--------------------------------------------------------------------------------------------------------

给程序传参：python xx.py 参数1 参数2 参数3 ...

import sys

**sys.argv #可以获得参数**

列表生成式: range(起始,结束,步长);range过大会占用大量内存空间，会内存溢出，只有在python2中会有。python3处理了,用range生成时，它会根据你的使用需要生成，你要一个它就生成一个。

a = [range(0,10)]

a = [i for i in range(1,18)]

a = [5 for i in range(1,5)] #生成4个5,结果是[5,5,5,5,5]

a = [i for i in range(10) if i%2 ==0]

a = [i for i in range(3) for j in range(2)] #得到[0,0,1,1,2,2]

a = [(i,j) for i in range(3) for j in range(2)] #得到[(0,0),(0,1)....]

可以写多个for

**列表在循环时进行删除会出现问题。循环时可以用一个新的列表来记录准备删除的数据，然后再用in来删除。**

模块搜索路径：先搜索当前路径，然后环境路径(sys.path).可以自己增加环境路径sys.path.append()

如果你导入了一个模块A,然后这个模块A进行了修改，那么你现在是不能使用到这些新功能的。你必须重新加载A模块

from imp import \*

reload(A)

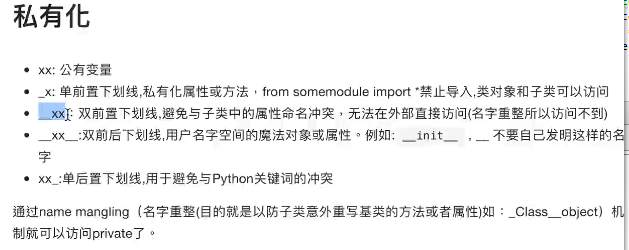
要避免循环导入

**is是不是同一个地址，==内容是不是相同**

**整型-5到256的is是相同的，放在同一个内存地址**

深拷贝import copy copy.deepcopy新建一内存地址复制内容

元组是不可变类型，你用copy也是指向同一个内存地址



以双下划线\_\_开头的变量是私有变量 ，外面不能访问。其实进行了名字重整，将私有变量的名字进行了更改\_类名\_私有属性。理论上不可访问，但实际上可以访问，但不要这样用。

class A(object):

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

\_\_name = 'coy'

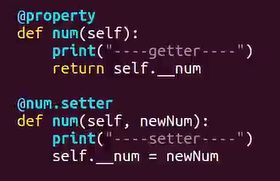
a = A()

print(a.\_\_name) #报错

a.\_\_name = 'kk' #但是可以给实例增加属性，增加一个同名的\_\_name

from 模块 import \* 导入模块后，被导入模块中的以下划线的变量是不能变导入的，不能被访问的。

import 模块时，模块中的下划线变量是可以访问到的。



外面直接使用num.

跟 num = property(get,set)一样的效果。只是@num.setter 中的num是外面要用的名字

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**

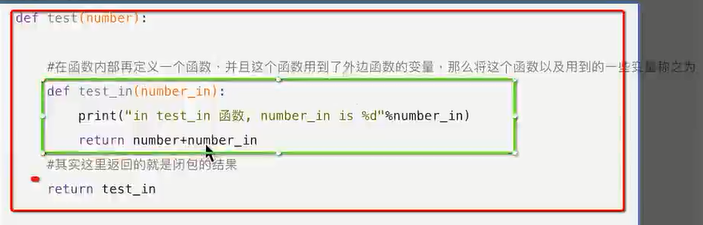
**验证可不可以迭代**

**from collections imoprt Iterable**

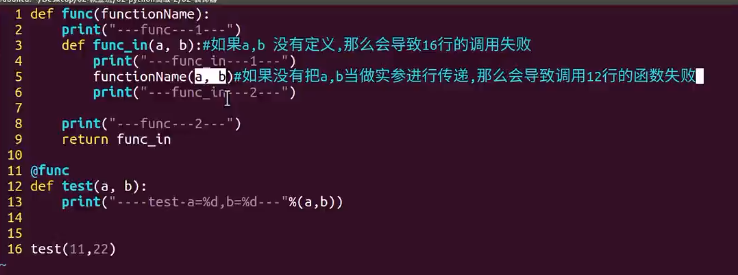
**isinstance(对象, Iterable)**

**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.**

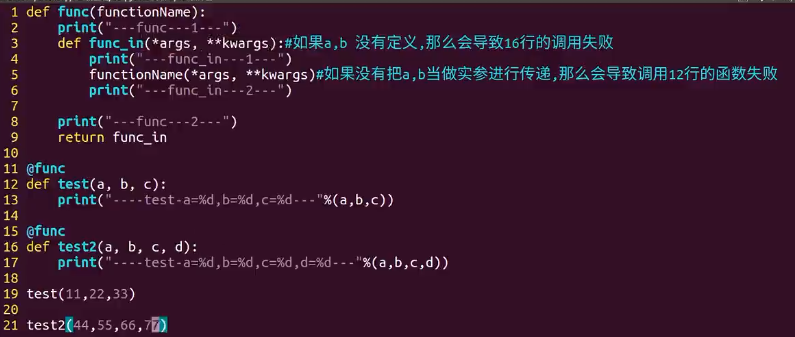
**转换成迭代器 b = iter(对象) 使用 next(b)**







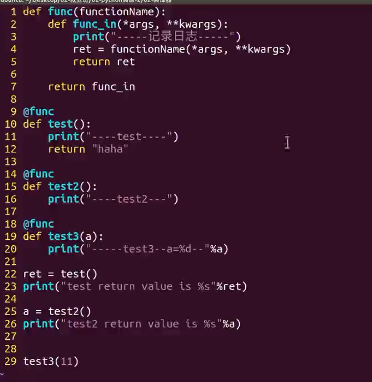
加装饰器时，如果原函数有几个参加就需要在装饰器中定义相应的参数。



可以使用可变参数和字典参数进行不定参数的传递。



带有返回值的装饰器



通用的装饰器，可以传多个参数，可以不带参数，可以有或没有返回值

装饰器带参数，在原有装饰器的基础上，设置外部变量。可以通过参数来做一些逻辑处理，可以对不同的被修饰者进行一些处理。



print(locals())

print(globals())

可以查看到局部变量和全局变量

python使用LEGB的顺序来查找一个符号对应的对象

locals(当前所在命名空间函数，模块，函数的参数也属于命名空间内的变量)->enclosing function(闭包)-->globals-->builtins

python是动态语言

静态语言运行之前要先编译

运行过程中给对象绑定（添加）属性。

class Person(object):

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

self.name = 'kk'

a = Person()

a.age=20 #给实例动态增加属性，原来类中没有

给类增加属性属于整个类，每个实例都会含有此属性

Person.addr = 'sz'

也可以对对象和类增加方法

**给实例增加方法，不能直接增加，要先引入types包**

import types

def eat(self):

print(self.name + 'eat')

a.eat = types.MethodType(eat, p1)

给实例增加方法，不能直接增加，要先引入types包

import types

def eat(self):

print(self.name + 'eat')

a.eat = types.MethodType(eat, p1) or types.MethodType(eat, p1)

增加静态方法：

@staticmethod

def test():

pass

P.xx = test #P是类名

增加类方法：

@classmethod

def printNum(cls):

pass

P.printNum = printNum #P为类名

**防止给对象增加属性：**

**class Person(object):**

**\_\_slots\_\_=("name", "age") #只允许给对象添加name和age,其它不行。**

-------------------------------------------------------

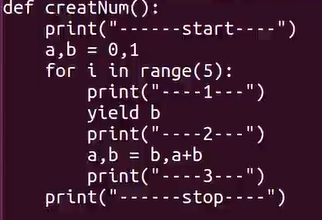
生成器：一边循环一边计算生成。

如果想要一个很大的列表还不能占用很大内存，可以使用生成器。不会先生成，用的时候再生成。保存计算方式，用的时候实现。

**生成器的实现方式:**

**b = (x\*2 for x in range(10)) #第一种生成方式，把列表的[]中括号改成括号(),使用next(b)获得下一个值**

**在for循环中使用yield,只要加了yield的函数都变成了生成器，调用要用next()**

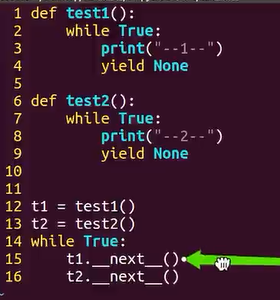


yield 的作用就是把一个函数变成一个 generator，带有 yield 的函数不再是一个普通函数，Python 解释器会将其视为一个 generator.在 for 循环执行时，每次循环都会执行 函数内部的代码，执行到 yield b时，函数就返回一个迭代值，下次迭代时，代码从 yield b的下一条语句继续执行，而函数的本地变量看起来和上次中断执行前是完全一样的，于是函数继续执行，直到再次遇到 yield。



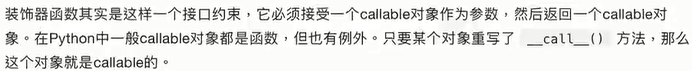
a.\_\_next\_\_()和next(a)是等价的

send()方法也可以有\_\_next\_\_的功能，让yield往下执行，如果send有传参数的话，会将参数给到yield的返回值 temp = yield b ，就是将参数传给temp,你可以对temp进行操作。如果没有用send ,yield b的返回值 是None



使用yield多任务程序

----------------------------------------------



如果一个类里面实现了\_\_call\_\_方法，那么可以直接对实例对象进行调用。如：

class Test(object):

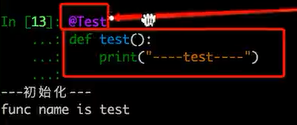
def \_\_call\_\_(self):

print("----test----")

t = Test()

t() #调用实例，可执行\_\_call\_\_方法







--------------------------------------------

元类

类就是一组用描述如何生成一个对象的代码段。类同样也是一种对象。

type可以动态创建类,不过这种做法不好 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))

class Test:

pass

Test2 = type("Test2",(),{}) #创建了一个class类

t2 = Test2()

最元始创建类的就叫元类

创建带有方法的类.1.先定义一个函数

def go():

pass

Test3 = type("Test3",(), {"go":go})

-------------------------------------------------------------



元类决定了将来要创建的类是什么样。

对象.\_\_class\_\_ 可以得到对象的类型，可以看到是什么(type或创建的类)创建的这个类型

一般不用type创建类

\_\_metaclass\_\_属性，你可以在定义一个类的时候为其添加\_\_metaclass\_\_属性，如果你这样做了，python就会用元类来创建类Foo.

class Foo(object):

\_\_metaclass\_\_ = something... #用 something去创建类，可以是函数，这个函数必须有3个参数(future\_class\_name,future\_class\_parents, future\_class\_attr)

...

----------------------------------------------------------------------------------------

垃圾回收：

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。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。超过[-5,257]创建的都有各自的内存空间。[-5,257]类似放在小整数池中。python一运行时就创建了。

intern机制:

创建相同的字符串数值时，它们共用一份数据(内存空间一样)。

python里每一个东西都是对象。

引用计数：简单，实时性：一旦没有引用就释放空间。引用计数解决不了循环引用的问题。

python链表查找,按一个方向找叫单链表，也有双链表找回去，跟java的根搜索算法查找一样。程序把所有的引用关系看作一张图，从一个节点GC ROOT开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点。

python中的对象一旦没有被使用，就立马释放空间。

隔代回收

查看引用数 sys.getrefcount(对象); gc.disable()关闭回收

显示垃圾回收 gc.collection()

\_\_getattribute\_\_属性拦截器



在\_\_getattribute\_\_和\_\_setattribute\_\_里禁止使用self.xxx不然会出现列循环

**xrange比range节省空间**

map函数会根据提供的函数对指定序列做映射

map(函数,列表)

#函数需要一个参数

ret = map(lambda x:x\*\*2,[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]) #把1给了x,把4给了y

#结果为:[5,7,9]

返回的结果主要看函数返回什么样的类型，可以返回元组的列表。

filter(函数,列表) 将列表过滤，filter(None, 列表)不过滤全取。

reduce函数会对参数序列中元素进行累积

reduce(lambda x,y :x+y,[1,2,3,4]) 把1给x,2给y..

set集合a,b求交集a&b，并集a|b，差集a-b,对称差集b^a

import functools 常用的工具包



偏函数，默认给了1,2,3不管如何传都有1，2，3





python -m pdb som.py #开启调试

l-显示当前的代码；n-向下执行一行代码；c-继续执行代码；b-break添加断点（b 9第9行断点）；clear删除断点；p打印一个变量的值；a-args打印所有的形参数据；q-quit;r-return快速执行到最后一行

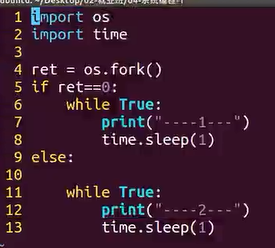
python编码风格 pep8 编码规范

并发：一个cpu只能执行一个任务，当任务多于cpu时就是并发。

并行：2个cpu，各个cpu同时执行一个任务，那么这2个任务就是并行。

多任务：PYthon的os模块封装了常见的系统调用，其中就包括fork,可以在Python程序中轻松创建子进程。只在linux有用。

ret = os.fork() 会创建一个新进程，那么就是一个主进程，一个子进程。子进程的返回值是ret=0,主进程的返回值是ret>0



在ret=os.fork()这里创建子进程，父进程和子进程都是从这里开始执行。

os.getpid()是获取当前进程ID；os.getppid()得到父进程id

如果在print时不想换行，可以print('hello',end='')用end来代替换行。

多任务，类似于多线程。

os.fork()创建子进程后，所有有全局变量，局部变量各自有一份，互不干扰。没有数据共享。进程之间没有沟通的桥梁。必须用特殊方式来沟通。

fork炸弹

while True;

os.fork()

-------------------------------------------------------------------------------

如果你打算编写多进程的服务程序，Unix/Linux无疑是正确的选择，由于windows没有fork调用。由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。multiprocessing模块就是跨平台版本的多进程模块。

from multiprocessing import Process

p=Process(target=函数名,args=('xx',)) 'xx'元组参数

p.start() #让进程开始执行代码

以后写多任务程序，可以把fork忘掉，就用这个multiprocessing,因为它是跨平台的。

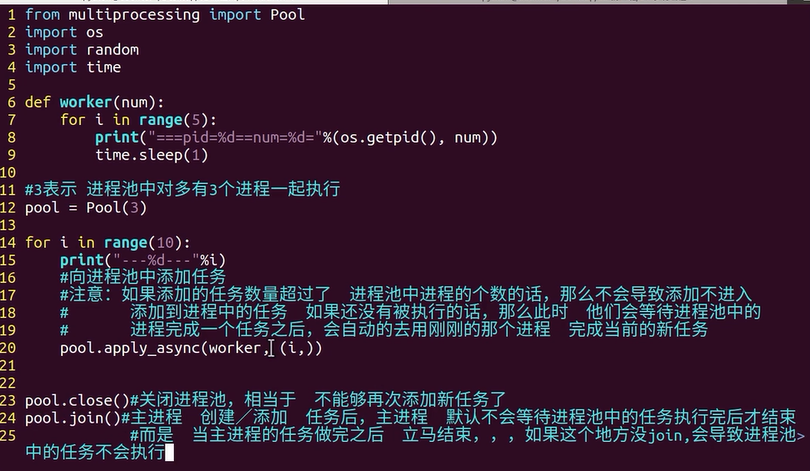


multiprocessing主进程会等待子进行运行完才结束。

p.join() #阻塞,等待p执行完成再往下执行。

p.terminate()不管任务是否完成，立即结束

可以用一个类继承Process这样也是创建一个子进程。



进程池Pool

from multiprocessing import pool

po=Pool(3)#定义一个进程池，最大进程数为3

po.apply\_async(函数,(参数,)) #Pool.apply\_async(要调用的目标,(传递给目标的参数元组,))

Pool.apply()是阻塞任务，上一个任务完成后再执行下一个

进程间通信 Queue 提供了同步的，线程安全的队列类FIFO,LIFO.可以用来模拟生产者和消费者模型。

from multiprocessing import Queue

q = Queue(3) 最多只能存3条信息

如果要使用进程池中通信，需要使用Manager().Queue()#使用Manager中的Queue来初始化

---------------------------------------------------

多线程

from threading import Thread

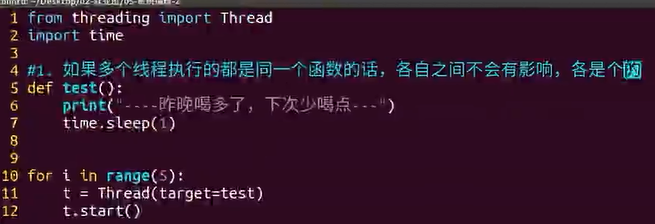
t1 = Thread(target=函数)

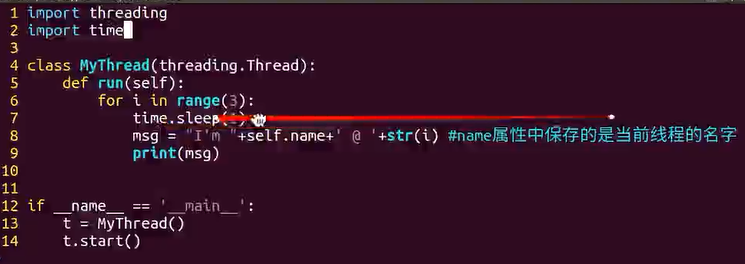
t1.start()

from threading import Thread

class MyThread(threading.Thread): #继承Thread实现自己的线程

def run(self):





子线程结束后主线程才结束。子线程结束后主线程才结束，主线程要为子进程收尸

主线程如果先结束，子线程就变成僵尸进程或孤儿线程

进程不管局部变量还是全局变量统统是各自进程的。

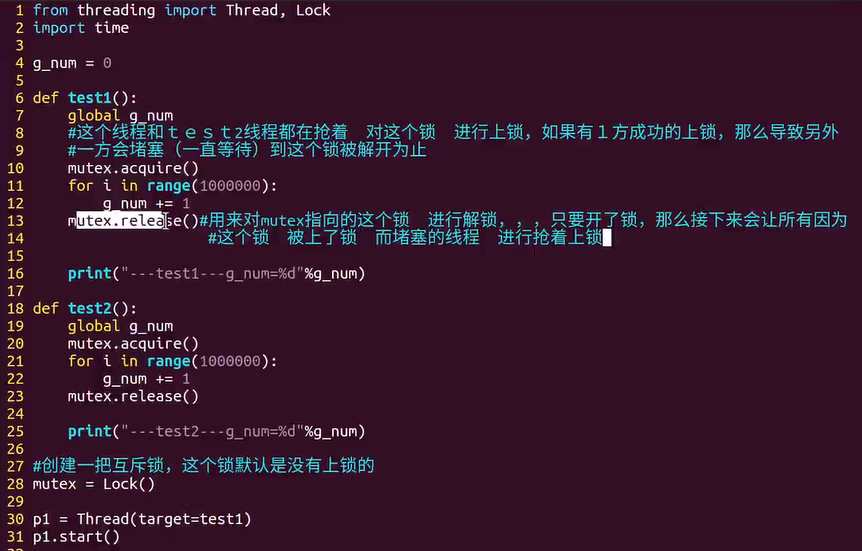
线程的全局变量是共享的，都可以进行操作。

**多个线程处于同一个进程里，所以共享全局变量。多进程不共享数据，要通过进程间的通信队列来通信Queue.**

多线程中把列表当做参数传进去也会被多个线程间共享。

------





mute.acquire()等待不占cpu,如果用while去判断就会占cpu资源。

同步就是协同步调，按预定的先后次序进行运行。如：你说完，我再说。“同”字应是指协同，协助，互相配合。不是指同时。

有顺序就是同步，没有顺序就是异步。

Queue阻塞队列就是用来给生产者和消费者解耦用的。

thread.local类似java的threadlocal



pool.apply\_async(func=test,callback=test2)

当test运行完了后，会调用回调函数test2,test的返回值会被当成参数传给test2

GIL全局解释器锁,用C语言来解决GIL问题。

假多线程

python规范

<https://google.github.io/styleguide/pyguide.html?showone=Naming#Naming>

这里是一个简单的函数，其参数和返回类型在注解中声明

**def** greeting(name: str) -> str:
**return** 'Hello ' + name

[Enum](http://python.usyiyi.cn/documents/python_352/library/enum.html#enum.Enum) callable有一个新参数**start**指定枚举值的初始值：

**Animal = enum.Enum('Animal', 'cat dog', start=10)**

'{0.platform:>10}'.format(sys) #格式化 platform表示sys的属性

一个函数内部的任何类型的赋值都会把一个名称或定为本地的。注意原处改变对象并不会把变量划分为本地变量，实际上只有对变量名赋值才可以。如果变量名L在模块的顶层被赋值为一个列表，在函数内部的像L.append(x)这样的语句并不会将L划分为本地变量，而L=X却可以。记住名称和对象之间的清楚的区分是有帮助的：修改一个对象并不是对一个名称赋值。

**def kwonly(a,\*,b,c)#代表传递参数时\*后面的参数必须是字典参数，如kwonly(1,c=3,b=2)，kwonly(c=3,b=2,a=1),如果是kwonly(1,2,3)就会报错。**

内置装饰器:

property, classmethod, staticmethod, functools.wraps,functools.lru\_cache

functools.lru\_cache 是非常实用的装饰器，它实现了备忘

（memoization）功能。这是一项优化技术，它把耗时的函数的结果保存

起来，避免传入相同的参数时重复计算。LRU 三个字母是“Least

Recently Used”的缩写，表明缓存不会无限制增长，一段时间不用的缓存

条目会被扔掉。

functools.singledispatch 装饰器，重载