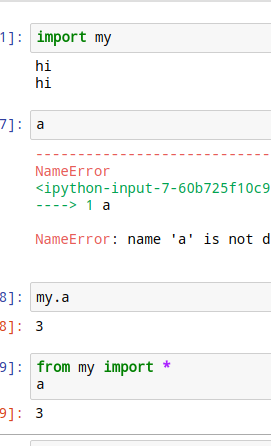
# Python语法

### Import 与from import

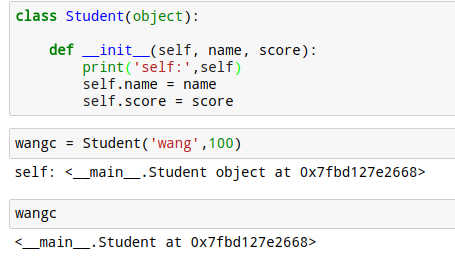
import只是包名字可见 ,用的时候要包名.\*\*

from 才是所有



### 面向对象

#### self:确实是对象本身,地址都是一个



#### \_\_xx,\_xx和\_\_xx\_\_

##### 特殊变量 如\_\_name\_\_ == ‘main’

\_\_name\_\_

\_\_author\_\_

\_\_doc\_\_



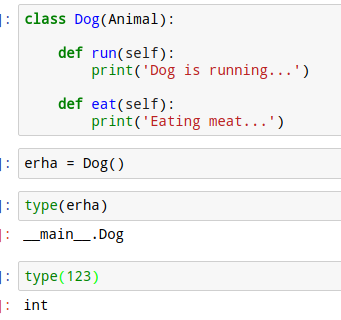
##### 假Private(\_××)



##### 真private(\_\_XX)

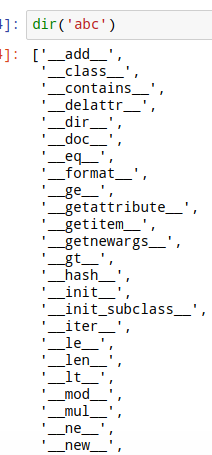


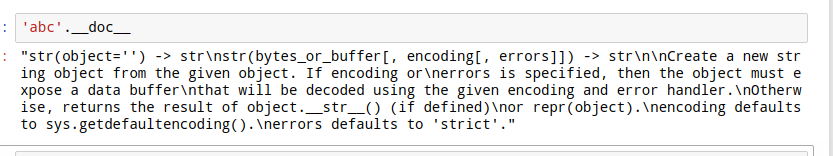
#### type() 和 isinstance()

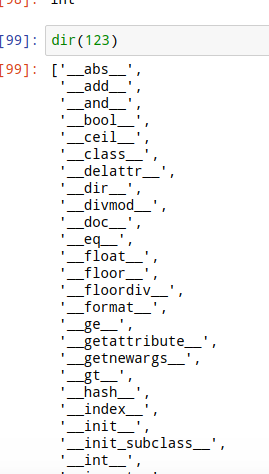




#### dir()返回所有属性和方法,再次验证一切皆对象！







#### hasattr() getattr() setattr()



#### 实例属性（self.xx）和类属性

###### 注意不要同名，否则会屏蔽



#### 动态绑定与\_\_slots\_\_限制

动态语言的特点





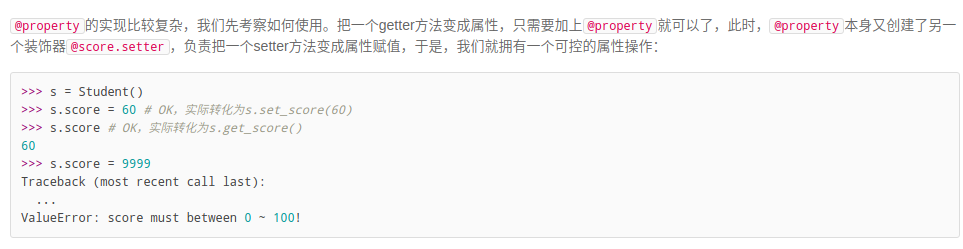
，

，

，

，

#### @property 方法变属性，看似暴露，但实际可能加了代码审查



#### 多重继承：继承多个类



#### 定制类使用特殊方法如\_\_str\_\_(),\_\_iter\_\_(),

<https://docs.python.org/3/reference/datamodel.html#special-method-names>



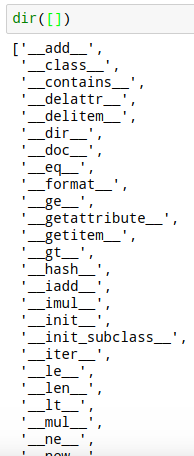
#### \_\_iter\_\_用于for循环

#### \_\_getitem\_\_:索引+切片\_





###### 栗子List 果然有 iter 和getitem 和len



#### \_\_call\_\_:把自身当函数用

### 错误

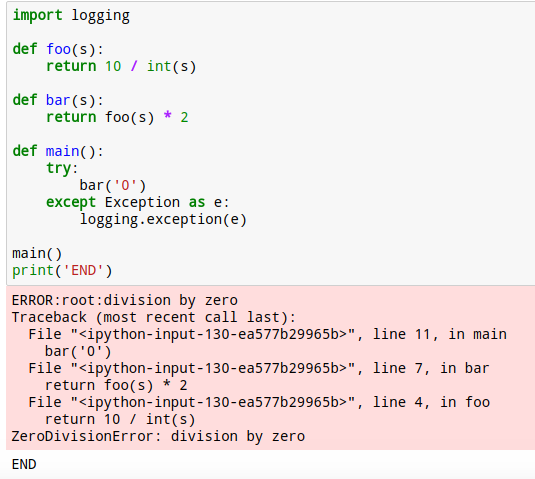
#### Try except else finally 捕捉错误



#### 报错调用栈



#### Logging记录错误，让程序保持正确运行



#### raise抛出自己定义的错误



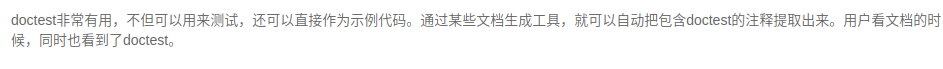
### 调试：print assert logging pdb单步调试，断点调试

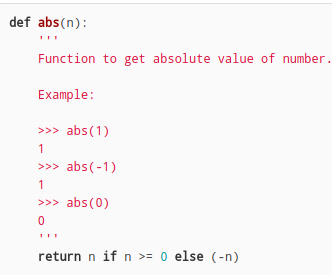
### 单元调试unitest模块



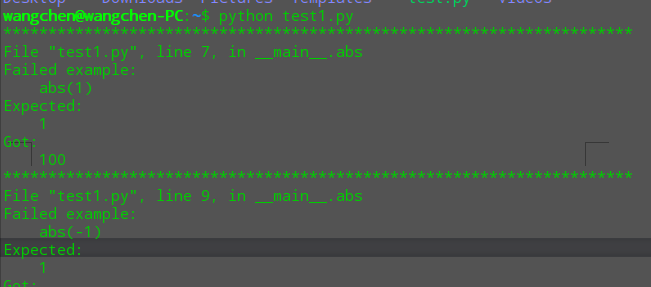
### 文档测试：从注释里面提取代码测试！测试，示例两不误！

###### 只能命令行，不能导入测



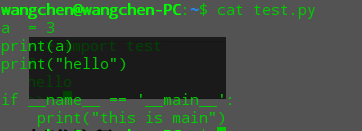






### 调用api???

### 导入模块 与 if \_\_name\_\_ ==’main’:



#### 命令行额外执行



#### 导入模块

发现，导入模块其实是把整个系统执行了一遍。但不执行\_\_name\_\_==’main‘中的



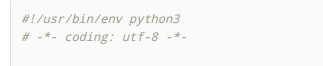
### 模块搜索路径（sys.path）



### sys.argv



### 标准注释

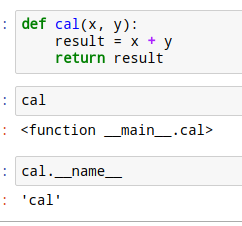


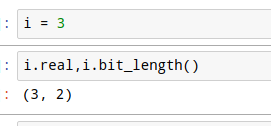


### 一切届对象

赋值就是建立一个对象的引用

例：





### 生成器与迭代器

#### 可迭代对象：能用for循环的！实现\_\_iter\_\_()的

\_\_iter\_获取迭代对象，

For循环只是语法糖

要获取迭代器。而迭代器又必须实现next()。迭代器本身是一次性使用的！要想多次使用，iter里面就不是self.

分一次性的，可多次性的。

一次性



多次性：



#### 迭代器（一次性）协议/类：next()函数，iter()也必须

<http://python.jobbole.com/81881/>

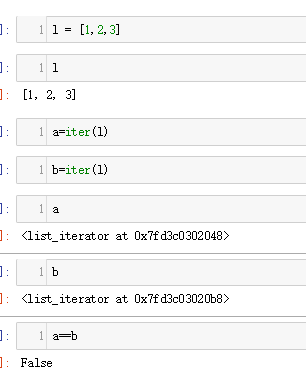


#### 多次性可迭代对象—iterable。\_\_iter\_\_返回迭代器。

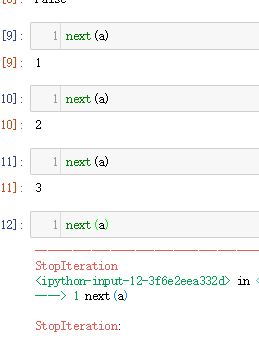


#### 例子说明：

两次的list不是一个

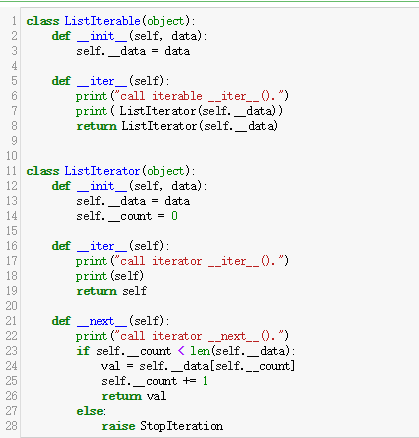


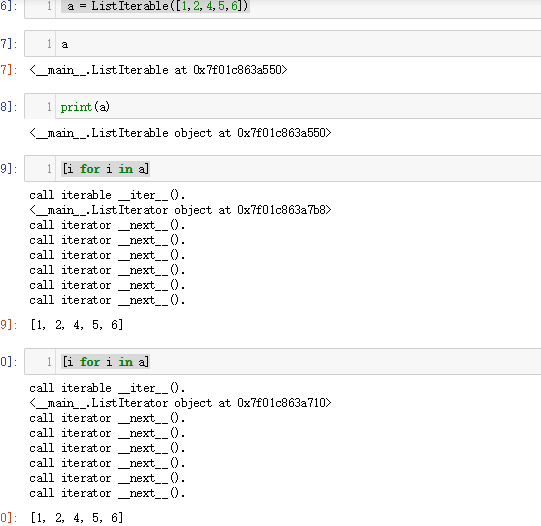
支持next()函数

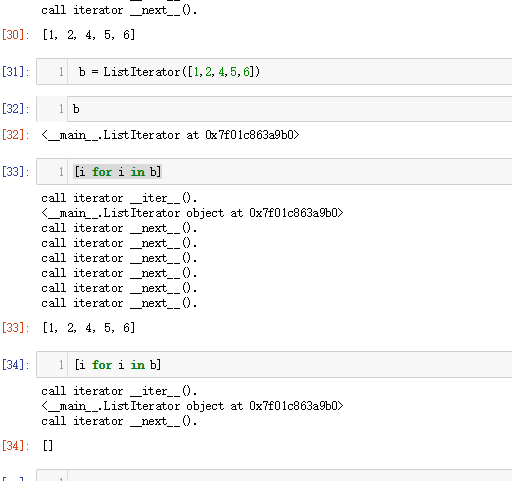


#### Iterable和iterator+for循环（完美例子，print大法）

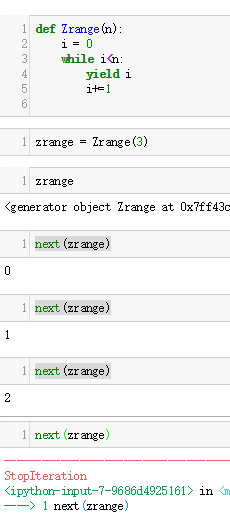
For循环 就是先iter ,再使用迭代器的next



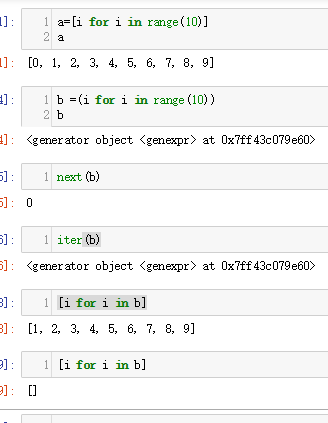




#### 生成器（函数）： 采用yield写法的特殊迭代器，所以也支持next

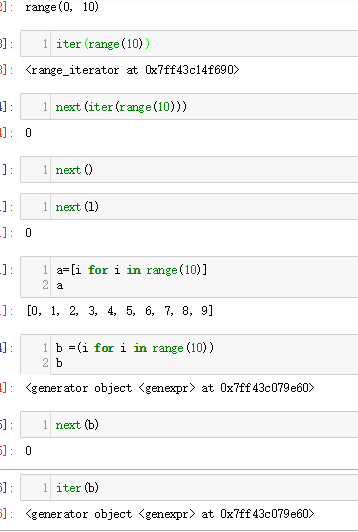


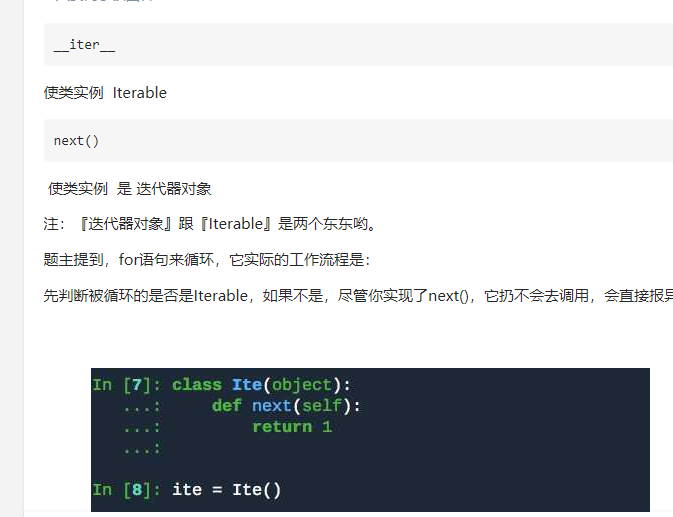
##### 既然是迭代器，那也不能重复使用



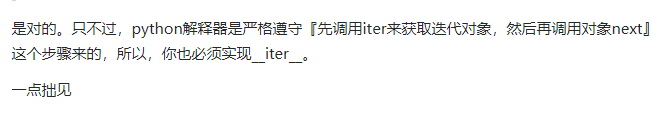
#### 生成器表达式（）和列表解析式[]

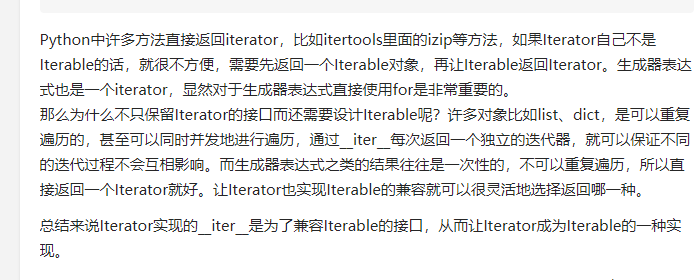
()是生成器，对内存更好。因为是生成器，蕴含yield，自然也是特殊的迭代器，支持iter和next









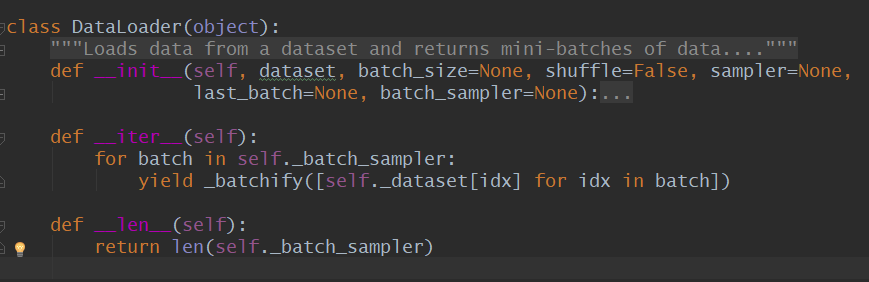


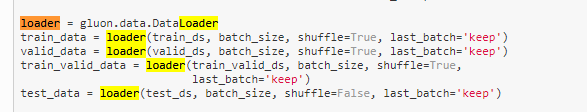
#### 在Mxnet源码中的应用

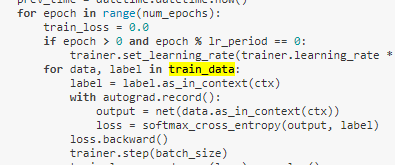
Mxnet中

中间这一坨应该是迭代器！，实际是生成器(yield写法的迭代器)

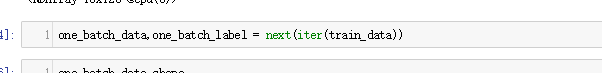
是iterable，所以可以重复使用



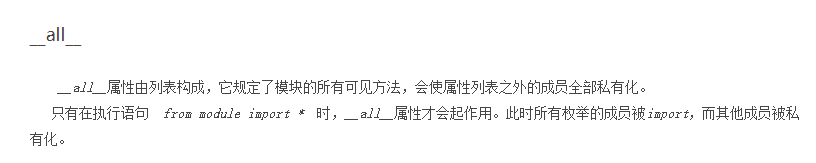


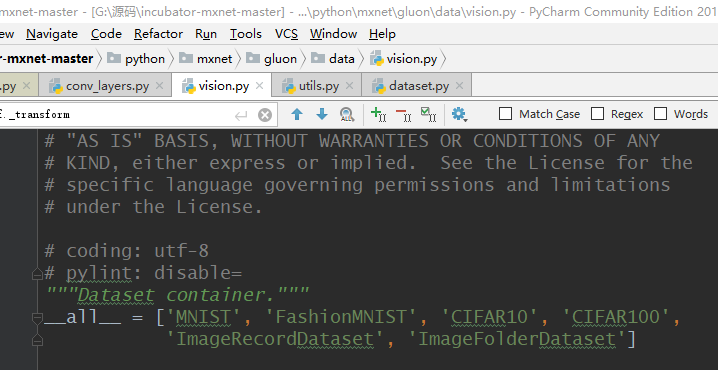


#### 在mxnet使用中拿一个数据例子

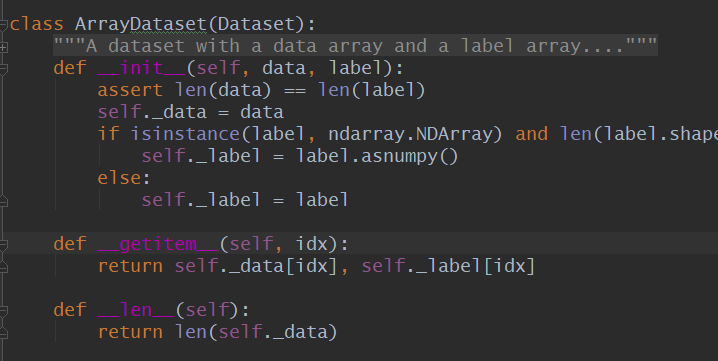


## \_\_all\_\_：可见方法

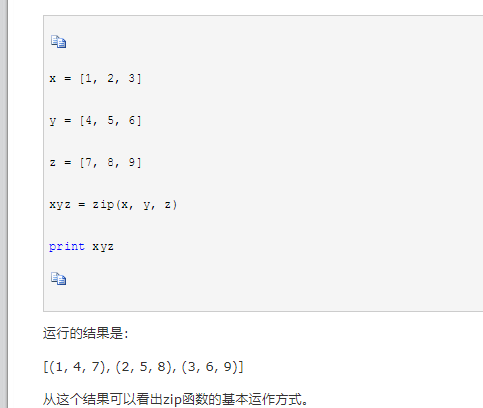




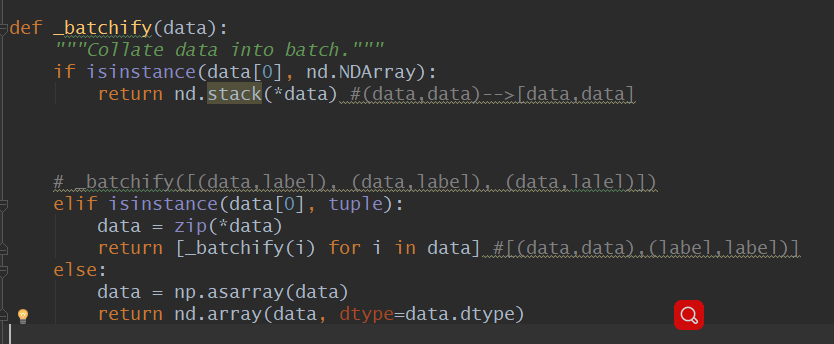
## \_get\_item\_\_:按索引拿元素

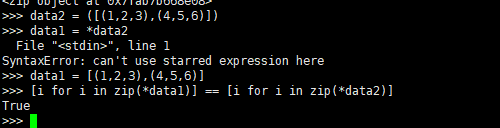


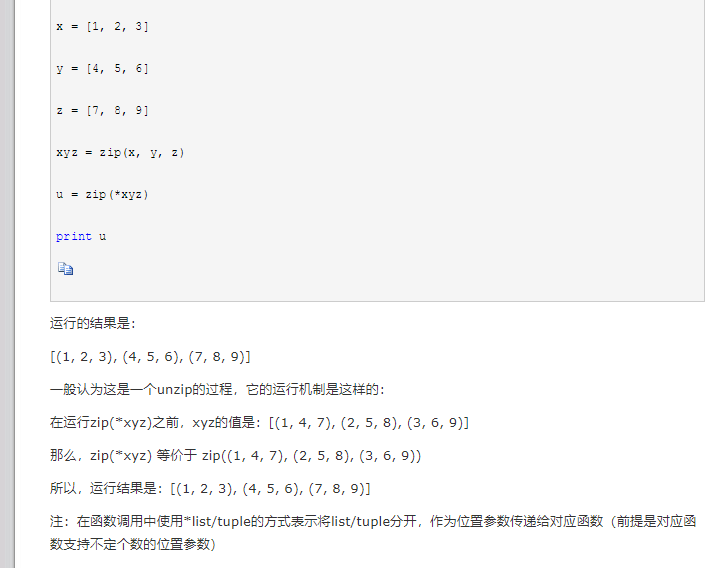
## zip():接受多个变量

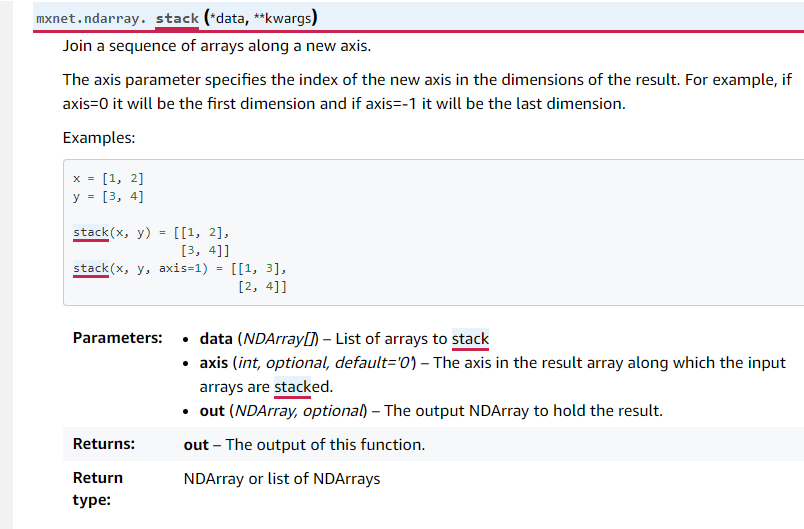
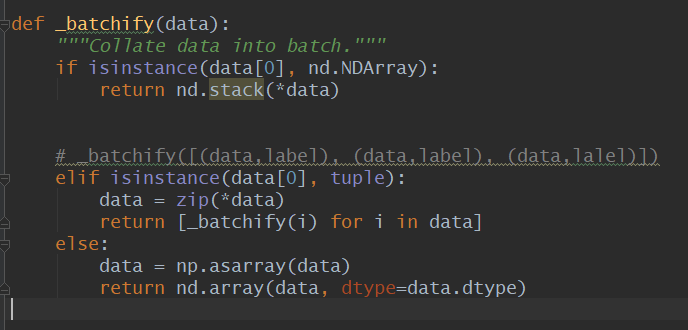


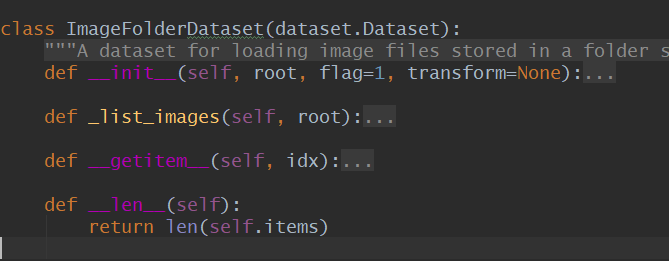
## zip(\*data) 接受一个变量 \*看作是把外面的层全打开，直到暴露分立的元素。

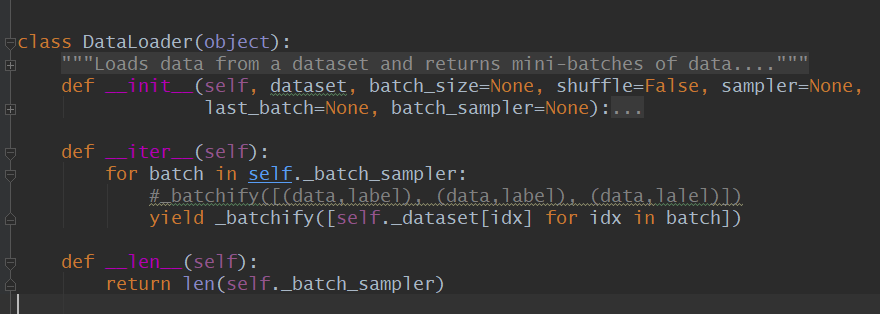




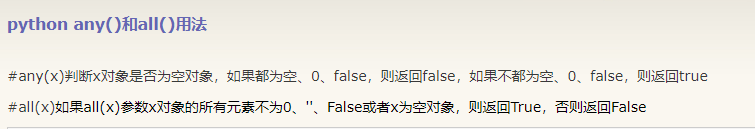






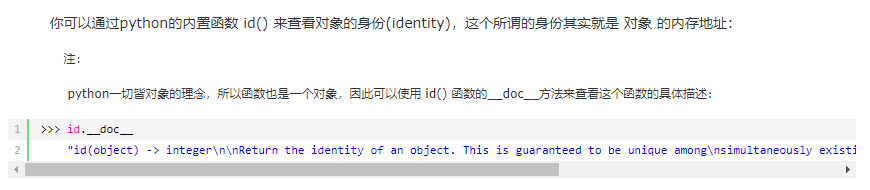


## any() all()



## 可变对象与不可变对象与global变量

##### Id()帮助透彻理解

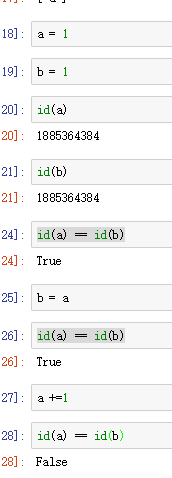




### 不可变对象（值变地址变）

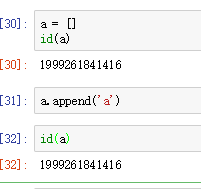
同一个值，同一个片内存。自身换值，地址也就不一样了。





### 可变对象（值本身上变地址不变）

##### 这里变是本身上变，不是另起炉灶的变

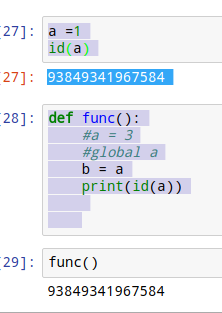


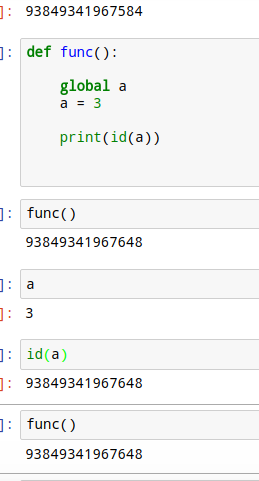
### Global变量

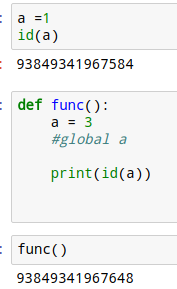
#### 不可变对象

写在函数外面的就是。

只是引用，不需要global。改变才需要global。因为有歧义

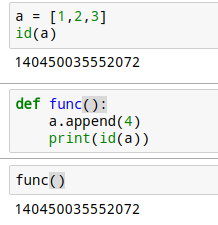
引用，地址相同

global了。内外地址就是一样的了。但是值变化了，不变对象，所以和最开始的地址不一样了、

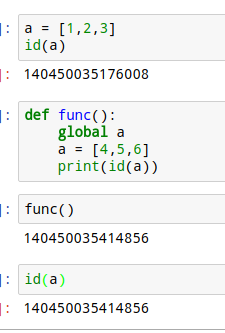
虽是同名，但是一个局部，一个全局。地址不同。

#### 可变对象

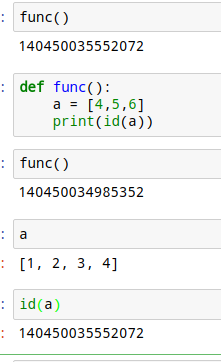
##### 本身修改全局，不用global,地址相同



##### 另起炉灶修改全局变量，要global

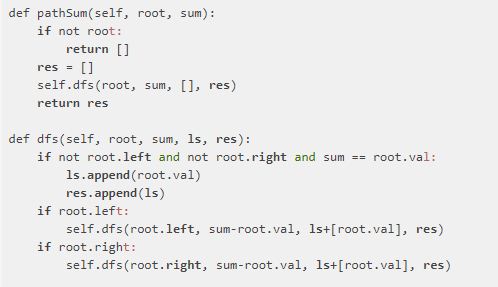
,最后地址相同。

##### 另起炉灶的同名局部变量

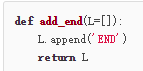


### Leetcode实例

这里res []发生了真正的变化。



## 可变对象作为函数参数(默认值会跟着变化)



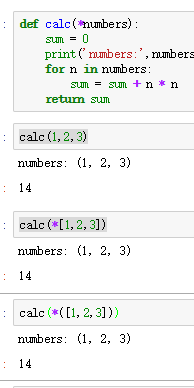


## 可变参数（\*args）\*在函数定义 和 函数使用中的区别

定义：表示将多个参数 组装成一个tuple;

使用时：讲tuple或list打开，暴露分立的参数 类比zip(\*data)

总结：\*这个位置，最后都是多个分立的参数

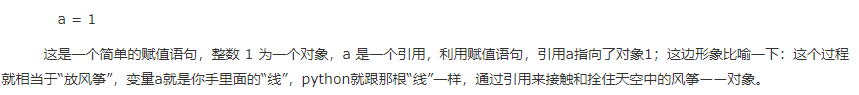


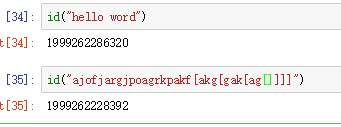


## 关键字参数\*\* （\*\*组装成dict,\*组装成tuple）

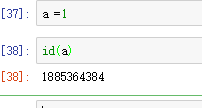
## 引用和对象：引用（变量名？指针？）

<https://www.cnblogs.com/ShaunChen/p/5656971.html>









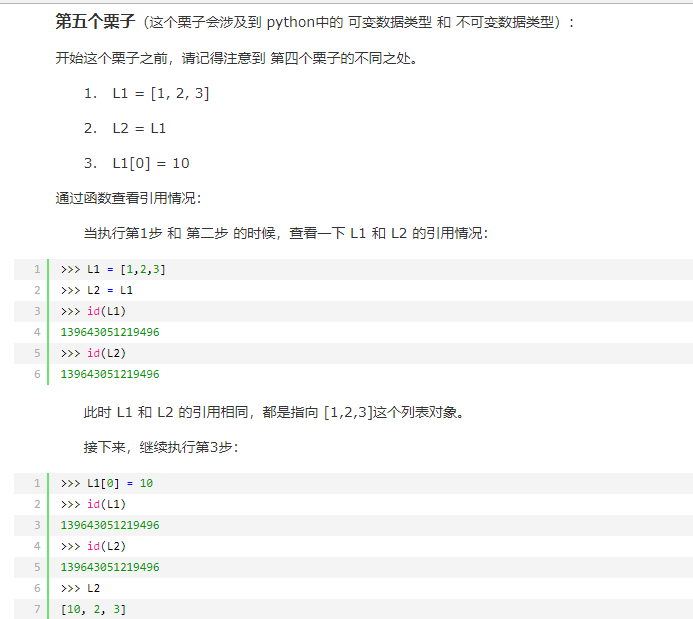


### 垃圾回收





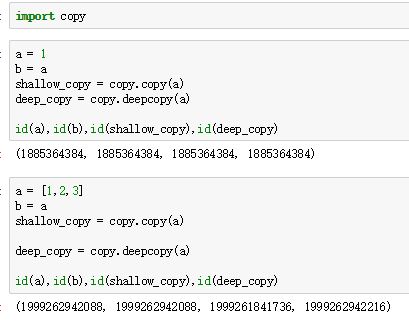
可变数据类型

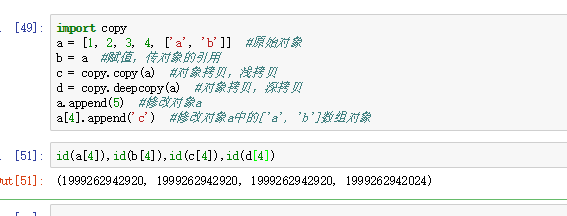


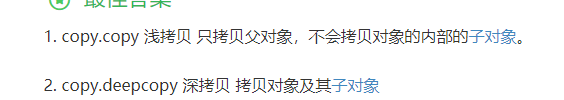
## 赋值、深拷贝与浅拷贝

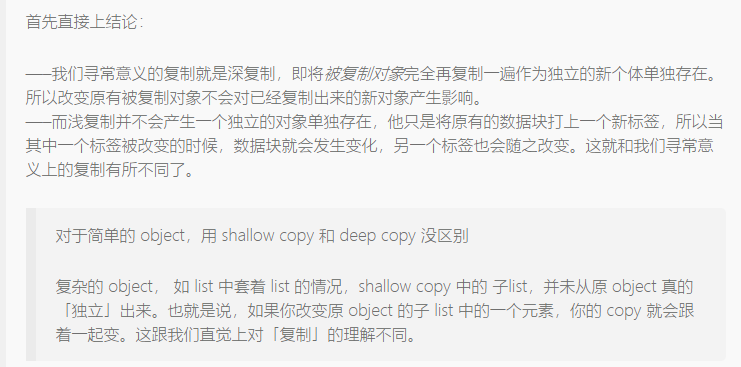
深拷贝：子对象也拷贝了

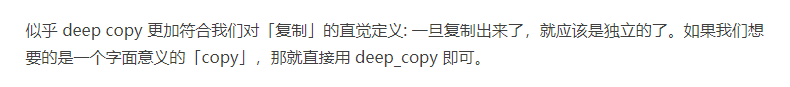
浅拷贝：只拷贝父对象，子对象不拷贝

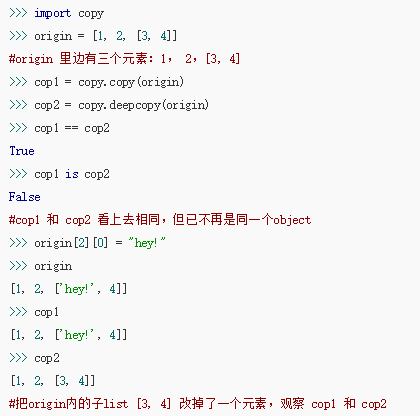


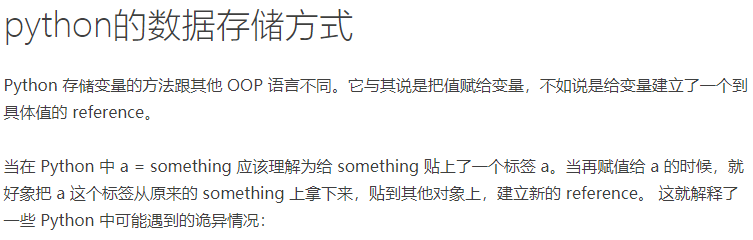


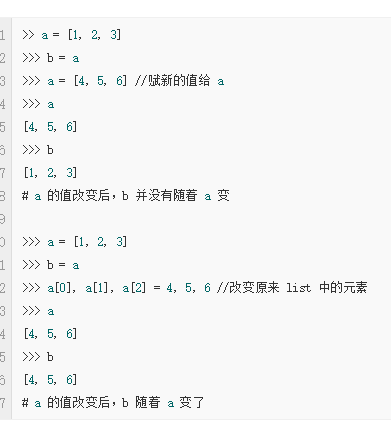


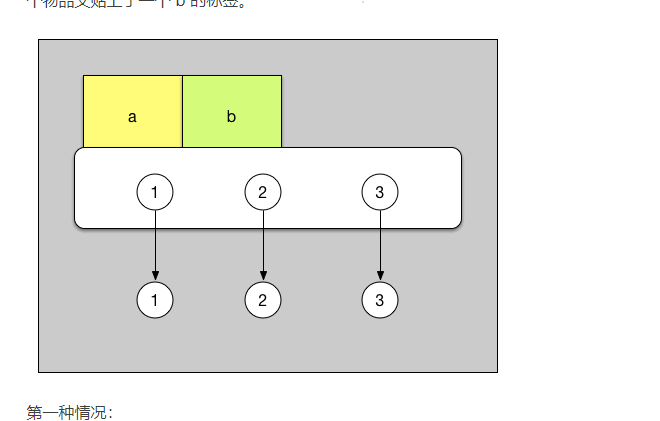


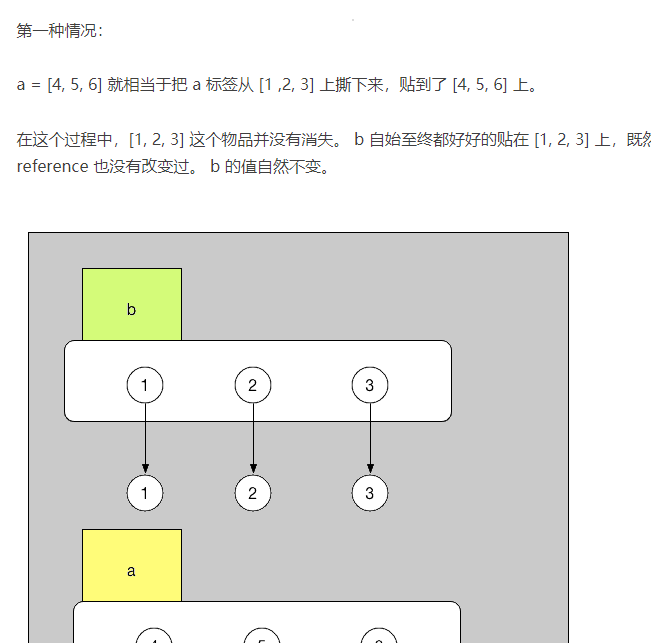












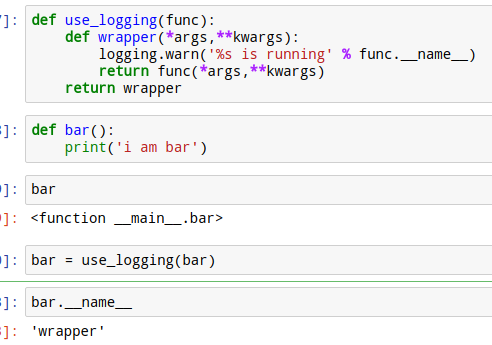


## 装饰器（给函数动态添加功能）与返回函数

是一个返回函数

新的函数实际上就是wrapper函数。





##### 带参数的，先算后两个，再@，就跟上面的道理一样了



## 协程

## 多线程

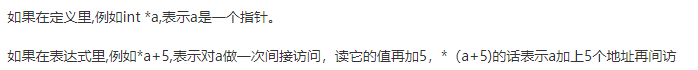
## 元类

## 错误，调试，测试

## 【】

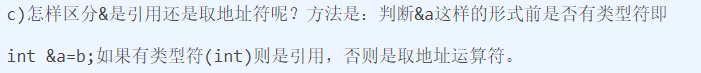
# C/C++

## 【\*运算符】int \*a vs \*a



## 【&运算符】int &a vs &a

#### 引用（别名）int & 区别：变量，指针

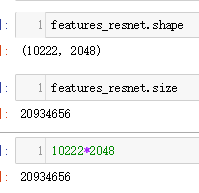


## 值传递(不可) 指针传递（可） 引用传递（可）

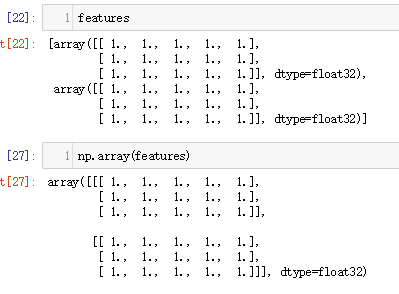
# 【各种环境下的数据】

# Numpy

## shape

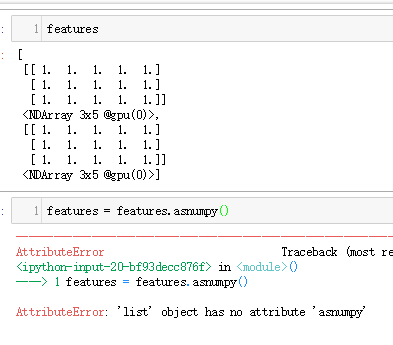


## 与list转换



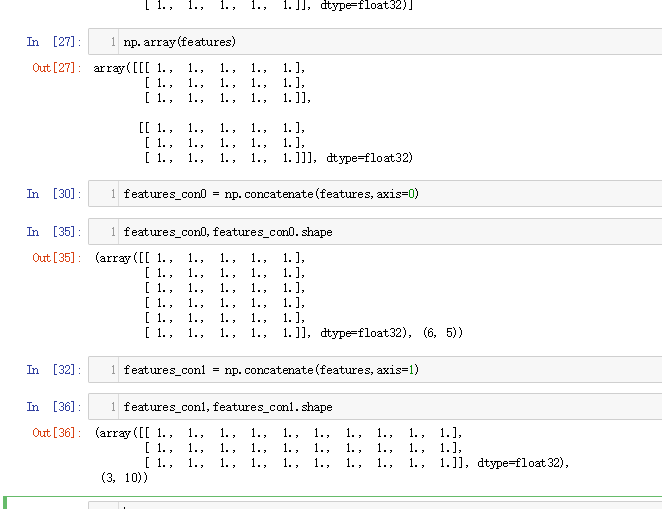
## 与ndarray互换





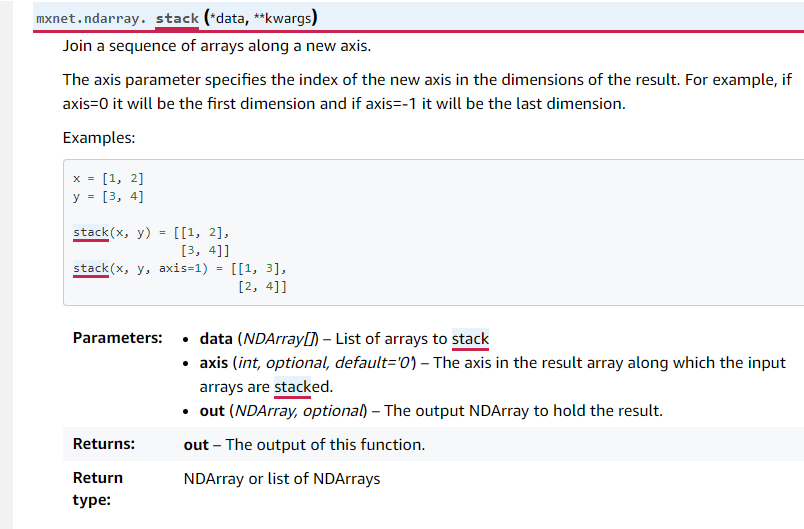
## np.concat轴（只针对np数据，若不是，要先asnumpy）

轴是谁就动谁



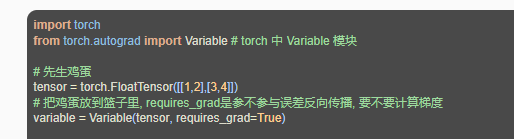
# Ndarray

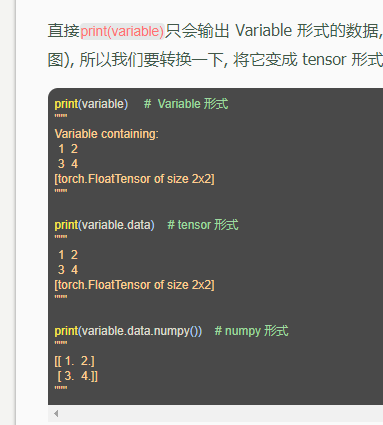
### stack



# Torch（Variable,torch,numpy）

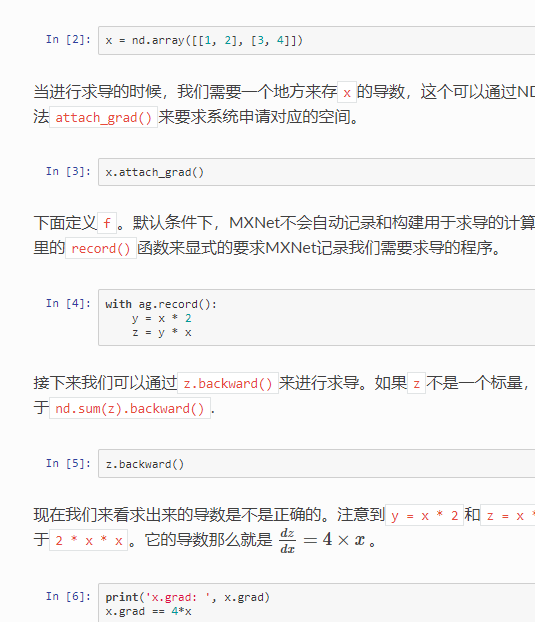
### 变量 与mxnet对比



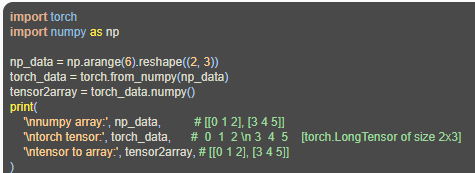


##### ! mxnet里面是with tf.record

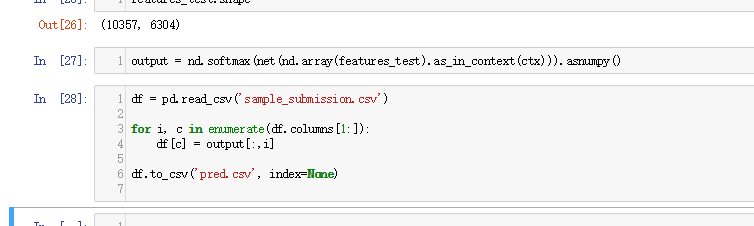
下面是mxnet



### 转换



# Pandas

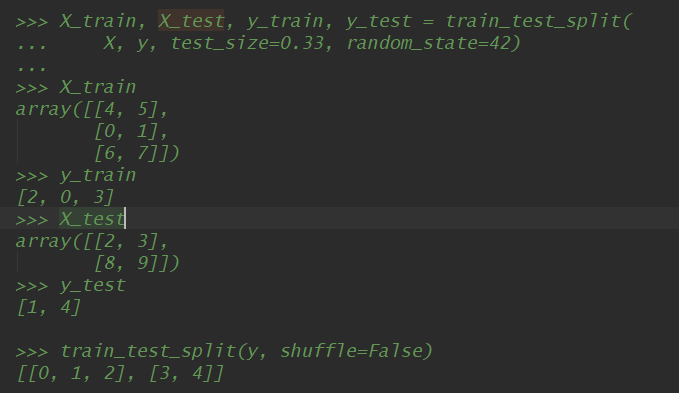
【】

# 【】

# SKlearn

## split 划分

尽量不自己去划分数据集！万一自己划分的有问题，也许特别影响效果、



# 其他模块

## 打印日志logging。

## 下载



但是不能看进度

-ubuntu wget就可以

## 文件夹操作

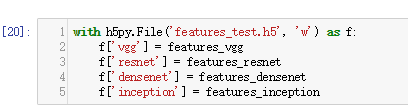
### -shutil.rmtree



## 画图

## H5py保存读取，不要用来存原始图片，巨慢！

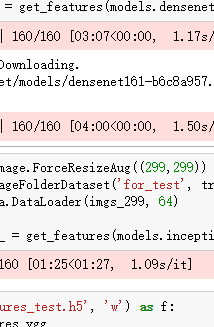
### File 类比 词典



## Tqdm

from tqdm import tqdm

进行时间和剩余时间，都能显示！



进行时间和剩余时间，都能显示！

