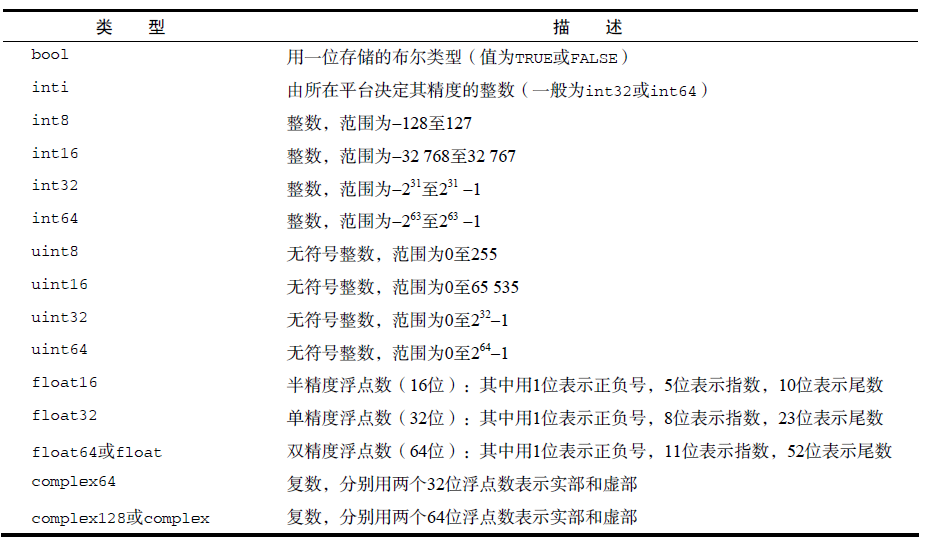
Python\_numpy函数解释

**1 创建多位数组**

在所有程序开头加上：import numpy as np

1.1 Numpy数据类型



完整的numpy数据类型可以使用np.sctypeDict.keys()找到。

数据类型名后面的数字表示单个数组元素在内存中所占用的位数。当然可以使用.dtype来显示数据的类型，而使用dtype.itemsize来显示单个数组元素在内存中所占用的字节数。如下例：

a = np.arange(12).reshape(3,4)

print(a)

print(a.dtype)

print(a.dtype.itemsize)

输出结果是：

[[ 0 1 2 3]

[ 4 5 6 7]

[ 8 9 10 11]]

int32

4 （这里的4表示字节数而不是位数）

1.2 区分Numpy生成的数组与Python自带的列表

首先需要说明一点的是：Python本身并不含有数组类型，只含有列表（list，也被称为向量），需要采用numpy才能生成数组（ndarray,在Python中表示成多维向量）。数组与列表在使用上有诸多不同，这里将对它们进行分析：

1.2.1 dtype与type的区别

1. 所有的数组均是由numpy库调用函数产生，包括一维数组和多维数组，而list列表是Python自带的，不用调用任和函数。数组元素的类型是np.ndarray类型，数组元素之间不含有逗号（仅在print时不含有），但是list数组类型并不完全相同，通常以逗号分隔。想要判断一个集合j是数组（ndarray）还是列表（list），可以使用type(j)来展示。

a = np.arange(12)

print(a)

print(type(a))

print(a.dtype)

b = list(range(12))

print(b)

print(type(b))

print(b.dtype)

[ 0 1 2 ..., 9 10 11]

<class 'numpy.ndarray'>

int32

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

<class 'list'>

Traceback (most recent call last):

File "H:/Python test/bc\_test/test\_0.py", line 16, in <module>

print(b.dtype)

AttributeError: 'list' object has no attribute 'dtype'

由此可以很明显地看出，type(j)对数组和列表均能使用，如果是数组，会返回<class 'numpy.ndarray'>，而如果是列表，则返回<class 'list'>。方法便捷、容易。

1. 一个数组中的函数类型均相同，即所有元素的类型只能是int8，int16等上面所述的类型的一种，因而可以使用dtype针对整个数组来显示数组中元素的类型；但是列表中的元素的数据类型可以不相同，它支持字符、数字、字符串甚至列表（即使用嵌套），所以只能使用循环并对每一个元素使用type来显示它的类型。

a = np.arange(12)

print(a)

print(type(a))

print(a.dtype)

c = [1, 234 , "abc", "7"]

for i in c:

print(type(i))

[ 0 1 2 ..., 9 10 11]

<class 'numpy.ndarray'>

int32

<class 'int'>

<class 'int'>

<class 'str'>

<class 'str'>

针对dtype与type的区别这里已经十分明显，type表现的是集合（无论是数组还是列表）中每一个元素的数据类型，dtype只能用于数组，表现的是数值类型。如果想分析列表的数值类型，只能针对每个元素调用type。

1. 仅有数组才有维度这一说法，表现的是数组的行、列以及其他通道数，可以使用array.shape。list列表没有维度，只能使用len(list)显示列表的长度。需要注意的是，数组的shape属性返回的是一个元组（tuple）,元组中的元素即为Numpy数组每一个维度的大小。见下例：

a = np.arange(12)

print(a)

print(a.shape)

b = list(range(12))

print(len(b))

c = [1, 234 , "abc", "7"]

print(type(c))

print(len(c))

[ 0 1 2 ..., 9 10 11]

(12,)

12

<class 'list'>

4

列表部分的解释比较清晰，但是我们上面可以看出，由于我们上面建立的数组是一维的，所以这里显示a的维数是（12，）而不是（12,1）。当时二维的数组时才能如此显示。这在进行数据计算时极容易弄混，我们将a转换成二维数据时显示如下：

a = np.arange(12)

print(a)

print(a.shape)

b = np.arange(12).reshape(1,12)

print(b)

print(b.shape)

[ 0 1 2 ..., 9 10 11]

(12,)

[[ 0 1 2 ..., 9 10 11]]

(1, 12)

在进行二维数组运算时就不会在产生数组维度不匹配的消息。

1.3 range与arange的区别

a = range(5)

print(a)

print(type(a))

[0, 1, 2, 3, 4]

<type 'list'>

import numpy as np

b = np.arange(5)

print(b)

print(type(b))

[0 1 2 3 4]

<type 'numpy.ndarray'>

首先，range是python自带的函数，无需引用函数库，而arange是numpy编程库里的函数，使用时要引用numpy函数库

其次，range返回的是一个数据组成的列表list，而arange返回的是numpy函数库中的ndarray数组。区别一方面可以从数值中看出来，a=[0, 1, 2, 3, 4]，每个数值之间是带逗号的，所以是列表list，b=[0 1 2 3 4]不带逗号，是数值计算属于numpy中的数组ndarray

这里list数组可以通过np.array()转换成ndarray数组：

f = np.array(a)

print(f)

print(type(f))

[0 1 2 3 4]

<type 'numpy.ndarray'>

这里你会发现f与b的数值以及类型是一样的，因此通过nump函数库中的ndarray()可以将列表list转换成ndarray数组

当然，若果转换成矩阵来进行相应数值计算的话，不必要从list转换成ndarray，直接转换成矩阵即可。

e = np.mat(a)

print(e)

print(type(e))

[[0 1 2 3 4]]

<class 'numpy.matrixlib.defmatrix.matrix'>

c = np.mat(b)

print(c)

print(type(c))

[[0 1 2 3 4]]

<class 'numpy.matrixlib.defmatrix.matrix'>

很明显两者是一样的，但这里需对c的元素进行相应的探究

print(c)

print(type(c))

print(c[0])

print(type(c[0]))

print(c==c[0])

[[0 1 2 3 4]]

<class 'numpy.matrixlib.defmatrix.matrix'>

[[0 1 2 3 4]]

<class 'numpy.matrixlib.defmatrix.matrix'>

[[ True True True True True]]

c是矩阵，包含的元素也相应的是一个矩阵，c可看成一个一行五列的二维矩阵，c[0]是第一行包含的矩阵元素，只不过也是一个二维矩阵

print(np.shape(c))

print(np.shape(c[0]))

（1, 5）

（1, 5）

c[0,3]表示的是第一行第三列的元素，如果只有一个输入元素，表示的是行，所以c[1]超出了范围range，包含的数据类型是numpy库中的numpy,int型

print(c[0,3])

print(type(c[0,3]))

print(type(c[1]))

3

<type 'numpy.int32'>

Traceback (most recent call last):

File "type.py", line 23, in <module>

print(type(c[1]))

File "C:\Users\Administrator\AppData\Roaming\Python\Python27\site-packages\num

py\matrixlib\defmatrix.py", line 318, in \_\_getitem\_\_

out = N.ndarray.\_\_getitem\_\_(self, index)

IndexError: index 1 is out of bounds for axis 0 with size 1

1.4 数组中元素序列的改变

1.4.1 翻转数组中所有的元素

对数组（无论是一维，二维等）的元素进行翻转，可以在相应的维度上进行：：-1。这一点在opencv上进行BGR通道翻转时十分必要。

a = np.arange(12).reshape(3,4)

print(a)

print(a[::-1,::-1])

print(a[:,::-1])

[[ 0 1 2 3]

[ 4 5 6 7]

[ 8 9 10 11]]

[[11 10 9 8]

[ 7 6 5 4]

[ 3 2 1 0]]

[[ 3 2 1 0]

[ 7 6 5 4]

[11 10 9 8]]

由上述的试验可知：想要对一个数组的元素进行彻底的翻转，只需要每一个通道上进行：：-1操作即可。如二维数组，可以进行a[::-1,::-1]

* + 1. 改变数组的维度（二维转化成一维）

将多维数组转换成二维数组可以使用reshape的方法，但是要将多维数组转化成一维数组的话，使用reshape会出错。这是为了减少运算，加强转化效率，可以使用一下几种方法：

1. 使用ravel()
2. 使用flatten()

两者在展现数组上式相似的，如下：

a = np.arange(24).reshape(2,3,4)

b = a.ravel()

c = a.flatten()

print(b)

print(c)

print(b.shape)

print(c.shape)

[ 0 1 2 ..., 21 22 23]

[ 0 1 2 ..., 21 22 23]

(24,)

(24,)

由此可以看出，三维数组被明显地改为了一维数组。有关两者的区别：

两者的区别在于返回拷贝（copy）还是返回视图（view），numpy.flatten()返回一份拷贝，对拷贝所做的修改不会影响（reflects）原始矩阵，而numpy.ravel()返回的是视图（view，也颇有几分C/C++引用reference的意味），会影响（reflects）原始矩阵。

a = np.arange(24).reshape(2,3,4)

b = a.ravel()

c = a.flatten()

b[1] = 100

d = a.flatten()

print(a)；print(b)；

print(c)；print(d)

[[[ 0 100 2 3]

[ 4 5 6 7]

[ 8 9 10 11]]

[[ 12 13 14 15]

[ 16 17 18 19]

[ 20 21 22 23]]]

[ 0 100 2 ..., 21 22 23]

[ 0 1 2 ..., 21 22 23]

[ 0 100 2 ..., 21 22 23]

由此可见，ravel会修改原先的数组，即使元数组的维度没有改变（因为没有赋值给元数组），但ravel后的元素值会传递给原数组。

* + 1. 数组矩阵的转置

在线性代数中，矩阵的转置是一个十分常见的操作，在多维数组中我们也可以进行相应的转置：

a = np.arange(12).reshape(3,4)

b = a.transpose() #或直接使用b = a.T

print(a)

print(b)

[[ 0 1 2 3]

[ 4 5 6 7]

[ 8 9 10 11]]

[[ 0 4 8]

[ 1 5 9]

[ 2 6 10]

[ 3 7 11]]

注：使用resize会致使直接修改原来的数组

a = np.arange(12).reshape(3,4)

print(a)

a.resize((4,3))

print(a)

若直接赋值b = a.resize((4,3))则会导致b = None。不知为何。

* 1. 查询数组属性的函数

1. ndim, 给出数组的维数，或者数组轴的个数。二维时为2，三维时为3等;
2. size，给出数组中所有元素的个数;
3. itemsize, 给出数组中单个元素在内存中所占的字节数;
4. nbytes, 查看整个数组所占的储存空间（字节数），等于size\*itemsize;
5. falt遍历多位数组的每一个元素，生成的是numpy.flatiter的数，不能整体显示，但可以遍历其中的每一个值，或每次遍历数个值。

a = np.arange(12).reshape(3,4)\*\*2

print(a.ndim)

print(a.size)

print(a.itemsize)

print(a.nbytes)

b= a.flat

print(b)

print(type(b))

c = a.flat[2]

print(c)

d = a.flat[[1,3]]

print(d)

a.flat[[1,3]] = 7

print(a)

2

12

4

48

<numpy.flatiter object at 0x000001B18D04FAE0>

<class 'numpy.flatiter'>

4

[1 9]

[[ 0 7 4 7]

[ 16 25 36 49]

[ 64 81 100 121]]

flat的赋值对原数组也产生了改变。

* 1. 数组与列表的相互转换

1.6.1 列表转换成数组

要先将一维的列表转换成数组，直接进行np.array()转化即可。

a = range(5)

print(a)

print(type(a))

b = list(range(5))

print(b)

print(type(b))

c = np.array(b)

print(c)

print(type(c))

print(b==c)

range(0, 5)

<class 'range'>

[0, 1, 2, 3, 4]

<class 'list'>

[0 1 2 3 4]

<class 'numpy.ndarray'>

[ True True True True True]

由此可见，直接使用range是无法生成list的，还要进行list化

* + 1. 数组转化成列表

列表的维度是1，所以要将数组转换成以一维数组后才能进行list的相关运算。一维转换的方法见1.4.2。下面介绍tolist函数可以将一维数组转换成列表。

a = np.arange(12).reshape(3,4)\*\*2

print(a)

b = a.flatten()

print(b)

print(type(b))

c = a.tolist()

print(c)

print(type(c))

d = b.tolist()

print(d)

print(type(d))

[[ 0 1 4 9]

[ 16 25 36 49]

[ 64 81 100 121]]

[ 0 1 4 9 16 25 36 49 64 81 100 121]

<class 'numpy.ndarray'>

[[0, 1, 4, 9], [16, 25, 36, 49], [64, 81, 100, 121]]

<class 'list'>

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]

<class 'list'>

由此可见，输出的c和d均是列表，但c是按行进行列表的，生成的是嵌套列表，而d才是我们需要的。所以进行数组的列表转换时一定要记得现将数组转换成一维的

* + 1. 列表转换为字符串

使用list命令和‘’.join(list)命令

a = “hello world”

b = list(a)

print(b)

c = ‘’.join(b)

print(b)

* + 1. 数组转化成指定的数据类型

使用astype函数。

a = np.arange(5)

print(a)

b = a.astype('float32')

print(b)

c = a.astype(float)

print(c)

d = a.astype(complex)

print(d)

e = a.astype('complex128')

print(e)

[0 1 2 3 4]

[ 0. 1. 2. 3. 4.]

[ 0. 1. 2. 3. 4.]

[ 0.+0.j 1.+0.j 2.+0.j 3.+0.j 4.+0.j]

[ 0.+0.j 1.+0.j 2.+0.j 3.+0.j 4.+0.j]

因此注意是否使用‘’标注的字符串。

* 1. 数组与字符串的组合与分隔

在文件处理中，读取或保存重要数据时，通常要对数据进行一系列的处理，以便于正确地处理数据，现在对常见的一些处理方式进行描述，以备后续使用。

1.7.1 数组的组合(hstack,vstack,dstack等)：

注意：无论是针对字符串还是数组，split分割后生成的都是list列表；但数组分隔后的元素认识数组，字符串分割后的元素仍是字符串。

这里介绍以不同的方式组合函数。首先创建两个数组：

>>> a = arange(9).reshape(3,3)

>>> a =

array([[0, 1, 2],

[3, 4, 5],

[6, 7, 8]])

>>> b = 2 \* a

>>> b

array([[ 0, 2, 4],

[ 6, 8, 10],

[12, 14, 16]])

1. **水平组合**

>>> hstack((a, b))

array([[ 0, 1, 2, 0, 2, 4],

[ 3, 4, 5, 6, 8, 10],

[ 6, 7, 8, 12, 14, 16]])

也可通过concatenate函数并指定相应的轴来获得这一效果：

>>> concatenate((a, b), axis=1)

array([[ 0, 1, 2, 0, 2, 4],

[ 3, 4, 5, 6, 8, 10],

[ 6, 7, 8, 12, 14, 16]])

1. **垂直组合**

>>> vstack((a, b))

array([[ 0, 1, 2],

[ 3, 4, 5],

[ 6, 7, 8],

[ 0, 2, 4],

[ 6, 8, 10],

[12, 14, 16]])

同样，可通过concatenate函数，并指定相应的轴来获得这一效果。

>>> concatenate((a, b), axis=0)

array([[ 0, 1, 2],

[ 3, 4, 5],

[ 6, 7, 8],

[ 0, 2, 4],

[ 6, 8, 10],

[12, 14, 16]])

1. **深度组合**

另外，还有深度方面的组合函数dstack。顾名思义，就是在数组的第三个轴（即深度）上组合。如下：

>>> dstack((a, b))

array([[[ 0, 0],

[ 1, 2],

[ 2, 4]],

[[ 3, 6],

[ 4, 8],

[ 5, 10]],

[[ 6, 12],

[ 7, 14],

[ 8, 16]]])

仔细观察，发现对应的元素都组合成一个新的列表，该列表作为新的数组的元素。

1. **行组合**

行组合可将多个一维数组作为新数组的每一行进行组合：

>>> one = arange(2)

>>> one

array([0, 1])

>>> two = one + 2

>>> two

array([2, 3])

>>> row\_stack((one, two))

array([[0, 1],

[2, 3]])

对于2维数组，其作用就像垂直组合一样。

1. **列组合**

列组合可将多个一维数组作为新数组的每一列进行组合：

>>> column\_stack((oned, twiceoned))

array([[0, 2],

[1, 3]])

对于2维数组，其作用就像水平组合一样。

1.7.2 数组的分隔（hsplit、vsplit、dsplit和split）

在NumPy中，分割数组的函数有hsplit、vsplit、dsplit和split。可将数组分割成相同大小的子数组，或指定原数组分割的位置。

1. **水平分割**

>>> a = arange(9).reshape(3,3)

>>> a

array([[0, 1, 2],

[3, 4, 5],

[6, 7, 8]])

>>> hsplit(a, 3)

[array([[0],

[3],

[6]]),

array([[1],

[4],

[7]]),

array([[2],

[5],

[8]])]

也调用split函数并指定轴为1来获得这样的效果：

split(a, 3, axis=1)

1. **垂直分割**

垂直分割是沿着垂直的轴切分数组：

>>> vsplit(a, 3)

>>> [array([[0, 1, 2]]), array([[3, 4, 5]]), array([[6, 7, 8]])]

同样，也可通过split函数并指定轴为1来获得这样的效果：

>>> split(a, 3, axis=0)

1. **面向深度的分割**

dsplit函数使用的是面向深度的分割方式：

>>> c = arange(27).reshape(3, 3, 3)

>>> c

array([[[ 0, 1, 2],

[ 3, 4, 5],

[ 6, 7, 8]],

[[ 9, 10, 11],

[12, 13, 14],

[15, 16, 17]],

[[18, 19, 20],

[21, 22, 23],

[24, 25, 26]]])

>>> dsplit(c, 3)

[array([[[ 0],

[ 3],

[ 6]],

[[ 9],

[12],

[15]],

[[18],

[21],

[24]]]),

array([[[ 1],

[ 4],

[ 7]],

[[10],

[13],

[16]],

[[19],

[22],

[25]]]),

array([[[ 2],

[ 5],

[ 8]],

[[11],

[14],

[17]],

[[20],

[23],

[26]]])]

1.7.3 字符串的分隔（split、os.path.split()）

在上文中我们看到了split用于数组的分隔，但是需要注意的是，split同样也可以用于字符串的分隔，只不过使用方法不太一样而已。

注意：无论是针对字符串还是数组，split分割后生成的都是list列表；但数组分隔后的元素认识数组，字符串分割后的元素仍是字符串。

split()：拆分字符串。通过指定分隔符对字符串进行切片，并返回分割后的字符串列表（list）

os.path.split()：按照路径将文件名和路径分割开

split()函数  
语法：str.split(str="",num=string.count(str))[n]

参数说明：  
str： 表示为分隔符，默认为空格，但是不能为空('')。若字符串中没有分隔符，则把整个字符串作为列表的一个元素  
num：表示分割次数。如果存在参数num，则仅分隔成 num+1 个子字符串，并且每一个子字符串可以赋给新的变量  
[n]： 表示选取第n个分片

注意：当使用空格作为分隔符时，对于中间为空的项会自动忽略

os.path.split()函数  
语法：os.path.split('PATH')

参数说明：

1. PATH指一个文件的全路径作为参数：
2. 如果给出的是一个目录和文件名，则输出路径和文件名
3. 如果给出的是一个目录名，则输出路径和为空文件名
4. 常用实例

>>> u = "www.doiido.com.cn"

#使用默认分隔符

>>> print u.split()

['www.doiido.com.cn']

#以"."为分隔符

>>> print u.split('.')

['www', 'doiido', 'com', 'cn']

#分割0次

>>> print u.split('.',0)

['www.doiido.com.cn']

#分割一次

>>> print u.split('.',1)

['www', 'doiido.com.cn']

#分割两次

>>> print u.split('.',2)

['www', 'doiido', 'com.cn']

#分割两次，并取序列为1的项

>>> print u.split('.',2)[1]

doiido

#分割最多次（实际与不加num参数相同）

>>> print u.split('.',-1)

['www', 'doiido', 'com', 'cn']

#分割两次，并把分割后的三个部分保存到三个文件

>>> u1,u2,u3 = u.split('.',2)

>>> print u1

www

>>> print u2

doiido

>>> print u3

com.cn

2) 去掉换行符

>>> c = '''say

hello

baby'''

>>> c

say

hello

baby

>>> print c.split('\n')

['say', 'hello', 'baby']

3) 分离文件名和路径

>>> import os

>>> print os.path.split('/dodo/soft/python/')

('/dodo/soft/python', '')

>>> print os.path.split('/dodo/soft/python')

('/dodo/soft', 'python')

1. 一个超级好的例子

>>> str="hello boy<[www.doiido.com]>byebye"

>>> print str.split("[")[1].split("]")[0]

www.doiido.com

>>> print str.split("[")[1].split("]")[0].split(".")

['www', 'doiido', 'com']

1.7.4 删除字符串的多余项(strip)

函数原型(strip)

声明：s为字符串，rm为要删除的字符序列

s.strip(rm) 删除s字符串中开头、结尾处，位于 rm删除序列的字符

s.lstrip(rm) 删除s字符串中开头处，位于 rm删除序列的字符

s.rstrip(rm) 删除s字符串中结尾处，位于 rm删除序列的字符

注意：删除后的结果认为字符串

1) 当rm为空时，默认删除空白符（包括'\n', '\r', '\t', ' ')

例如：

>>> a = ' 123'  
>>> a.strip()  
'123'  
>>> a='\t\tabc'  
'abc'  
>>> a = 'sdff\r\n'  
>>> a.strip()  
'sdff'

1. 这里的rm删除序列是只要边（开头或结尾）上的字符在删除序列内，就删除掉。

strip不能删除中间的字符串，比如”123456”,只能删除1开头的或者6结尾的，而像234,345,45这种字符序列就删除不掉。

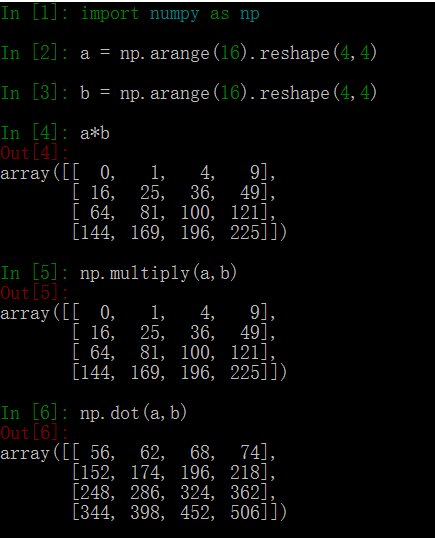
例如 ：

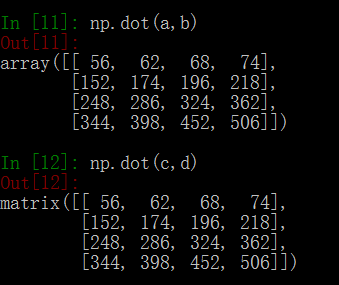
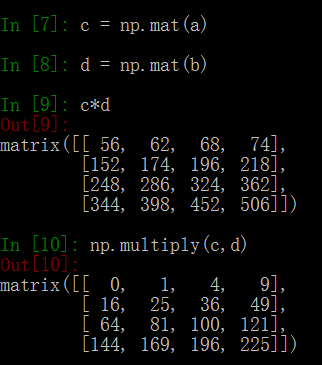
代码如下:

>>> a = '123abc'  
>>> a.strip('21')  
'3abc' 结果是一样的  
>>> a.strip('12')  
'3abc'

1.8. Python中数组的点乘和矢量乘的实现方式：

使用array时，运算符 \* 和multiply均用于计算数量积(点乘)，函数 dot() 用于计算矢量积(叉乘)。  
使用matrix时，运算符 \* 和dot()用于计算矢量积，函数 multiply() 用于计算数量积.。





1.9. Numpy中array,ndarray,nsarray的区别

* **array 与ndarray的区别：**

简单说 np.array只是一个便捷的函数，用来创建一个ndarray，它本身不是一个类。它返回numpy.ndarray这个类(class);

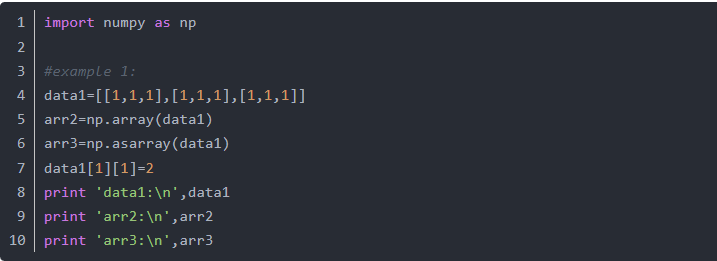
与numpy.array这个函数同类型的有： np.ones,np.zeros np.empty;

但是, 不建议使用np.ndarray(…)创建一个ndarray

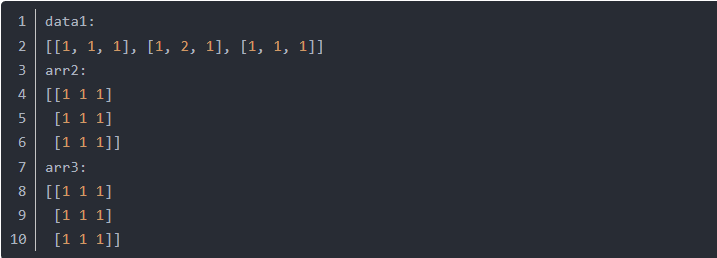
* **array 与nsarray的区别：**

array和asarray都可以将结构数据转化为ndarray，但是主要区别就是当数据源是ndarray时，array仍然会copy出一个副本，占用新的内存，但asarray不会。

比如：



输出：



可见当数据源是list列表时，array和asarray没区别



输出：



可见：当数据源是ndarray时, nsarrat不会copy数据

**2 文件阅读模式和常用转义字符**

**2.1 \t,**[**\r,\n,\r\n的区别**](http://blog.csdn.net/u010189070/article/details/50955493)

在Python的字符串中，通常会用到转义字符，现在对转义字符进行详细的解释。

\t是TAB键，相当于四个空格键，横向跳到下一制表符位置

\n是换行，英文是New line，表示使光标跳到下一行行首  
\r是回车，英文是Carriage return，表示使光标回到当前行的开始处

\r\n表示回车换行

我们在平时使用电脑时，已经习惯了回车和换行一次搞定，敲一个回车键（Enter），即是回车，又是换行(\r + \n)。

a = '\t\tab\nc'

print(a)

b = a.strip()

print(b)

c = 'sdff\r\nmpk'

print(c)

d = c.strip()

print(d)

e = 'sdff\rmpk'

print(e)

ab

c

ab

c

sdff

mpk

sdff

mpk

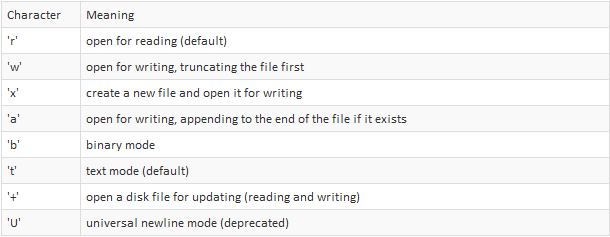
mpkf

**2.2 读取文件中的文件打开字符串的含义**

表格来源：[CSDN](https://blog.csdn.net/qq_38410730/article/details/80242624)

[](https://blog.csdn.net/qq_38410730/article/details/80242624)

Python函数库里是：



**3. 读取、存储和打包文件**

3.1 读取和存储文件

读取.txt文件的方式有两种，但根据实际的问题，需要进行不同的计算：

1. 使用loadtxt()读取TXT文件

此函数为numpy库中的函数，用于读取TXT文件以及CSV文件，这里仅分析读取TXT文件。

np.loadtxt( fname, dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0) 输出值是一个数组

fname: 文件名（包含file名称, str, or 文件的路径）,如果文件的扩展名是.gz或者是.bz2,则文件会首先进行解压，再返回字节字符串。

dtype：输出数组中元素的类型。默认为float类型，如果想要得到str类型，则应该使用dtype=‘str’, 如果使用结构化的数据类型的话，输出数组将是一维的，每一行的数据作为数组的一个元素，因而要求每一列列数是一致的。

comments: 注释（字符串或者是序列）， 尝试用#表示一个注释的开始。

delimiter：字符串形式，表示使用它来分类每一个值。默认为空格。

converters: 转换器，输入数据通常是一个字典，用来将某一个列数中的值转换为float行。比如：如果第0列表示时间的字符串，converters = {0: datestr2num}。当然，这个参数用来补齐某个位置缺失的值（相关用法见：genfromtxt）：converters = {3: lambda s: float(s.strip() or 0)}. 默认不开启。

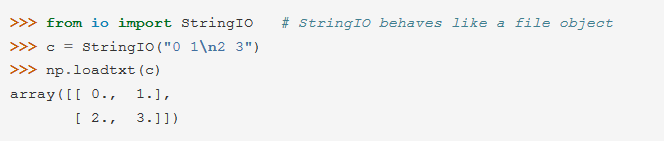
skiprows：表示读取TXT文档时，需要忽略的行数（比如第一行是标题，则从第二行读起，此时=1），默认为0.

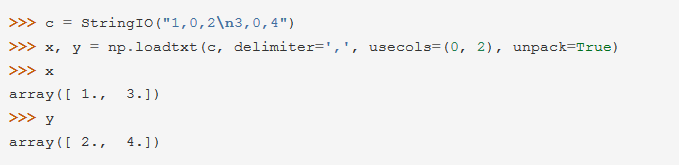
usecols：读取的列数，输入值是整数值或者是元组。表示读取文档中的第几个列。=0表示只读取第一列，=（1,4,5）时表示读取数组的第2,5,6列。默认是读取所有的列。

unpack：用于表示输出的值得个数，类型为布尔类型，当为None时输出值为一个数组，当为True时，TXT文档将会被读为结构化数组，x, y, z = loadtxt(...).，每一个输出值对应一个列数。简单来说，当unpack参数设置为True时，意思是分拆存储不同列的数据，即分别按列将数组赋予不同的变量。

ndmin: 表示输出数组应有的维度，表示该数组至少会被分为几维，否则数列将会被挤压。默认为0，1或者2时常见的合法值。

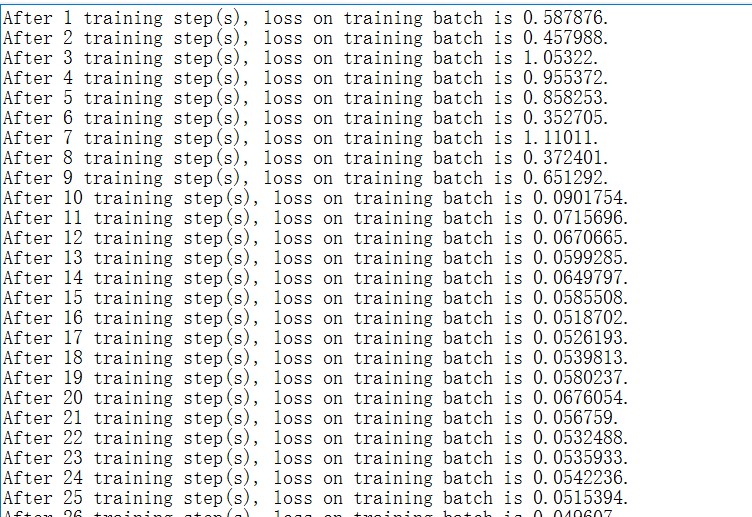
举例：





尤其注意第二个用法举例。

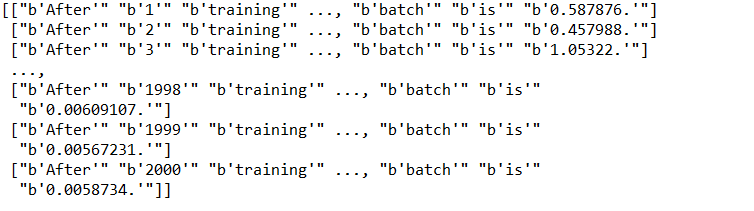
以本文中涉及到的一个shuju.txt为例进行分析，此分析可用于以后常见的数据处理中。



上述是shuju.txt文档中的内容，我们这里想读取每一行最后的损失值，画出损失值与训练次数之间的关系。这里的loadtxt的输出数组的dtype不能设置为float，因为数组中大部分是英文字母而不是数字。因此想要使用loadtxt，只能将类型设置成string，即：

b = np.loadtxt('shuju.txt', dtype='str')

输出的结果为：



由此可见：每一项为 b'0.0058734.'，与我们希望的数值‘0.0058734’相差较远，想得到这个数值又要进行一堆更改，较为麻烦。所以如果要是针对这种数字与字母、汉字相结合的，尽量采用别的方法，但loadtxt方法针对全是数字的文本有较好的分析能力。

1. 使用loadtxt()读取CSV文件

np.loadtxt( fname, dtype=<type 'float'>, comments='#', delimiter=’,’, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0) 输出值是一个数组

由此可见读取csv文件时，只需将delimiter设置为’,’逗号分隔符即可。

3）使用savetxt()保存TXT文件

np.savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n', header='', footer='', comments='# ', encoding=None)

fname: 文件名（包含file名称, str, or 文件的路径）,如果文件的扩展名是.gz或者是.bz2,则文件会首先储存为gzip压缩文件格式。

X:将要保存的数据

fmt:数据格式

delimiter: str, optional. 字符串或字符用来分隔列。

newling: str, optional. 字符串或字符用来分隔行。

header : str, optional. 将在文件开始时写入的字符串。

footer : str, optional. 将在文件结束时写入的字符串。

x = y = z = np.arange(0.0,5.0,1.0)

np.savetxt('test.out', x, delimiter=',') # X is an array

np.savetxt('test.out', (x,y,z)) # x,y,z equal sized 1D arrays

np.savetxt('test.out', x, fmt='%1.4e') # use exponential notation

3）使用savetxt()保存CSV文件

np.savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n', header='', footer='', comments='# ', encoding=None)

由此可见保存csv文件时，只需将delimiter设置为’,’逗号分隔符即可。

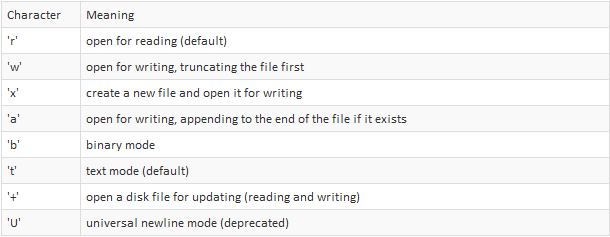
4）使用open()读取TXT文件

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)

打开文件并返回一个字符串流（stream）,我读取时是为str。失败时产生IOError。

file：文件可以是文本或字节字符串，给出要打开的文件的名称(如果文件不在当前工作目录中，还提供路径)，也可以是要包装的文件的整数文件描述符。(如果给定文件描述符，则在关闭返回的I/O对象时关闭它，除非将closefd设置为False。)

mode是一个可选字符串，它指定打开文件的模式。它的默认值是“r”，这意味着在文本模式下可以打开阅读。其他常见的值有“w”表示写入(如果文件已经存在，则截断它)、“x”表示创建和写入新文件，以及“a”表示追加(在某些Unix系统中，这意味着所有写入都将追加到文件的末尾，而不管当前的查找位置如何)。在文本模式中，如果没有指定编码，则使用的编码依赖于平台:调用local .getpreferredencoding(False)来获取当前的地区编码。(对于读取和写入原始字节，使用二进制模式，不指定编码)



open默认的的文件打开模式是rt,就是只读模式下读取文本文件，而不是二进制文件（这个要注意））,  The 'x' mode implies 'w' and raises an FileExistsError if the file already exists.

buffering : buffering是一个可选整数，用于设置缓冲策略。传递0以关闭缓冲(仅在二进制模式下允许)，传递1以选择行缓冲(仅在文本模式下可用)，传递一个整数n> 1以指示固定大小块缓冲区的大小。如果取负值，寄存区的缓冲大小则为系统默认。

encoding：是用于解码或编码文件的编码的名称。这应该只在文本模式下使用。默认编码依赖于平台，但是Python支持的任何编码都可以传递。有关受支持编码的列表，请参见codecs模块。

errors:错误是一个可选字符串，指定如何处理编码错误---这个参数不应该在二进制模式下使用。如果存在编码错误(the default of None has the same effect)，则传递“strict”来引发ValueError异常，或者传递“ignore”来忽略错误。(请注意，忽略编码错误会导致数据丢失。)参见编解码器文档。注册或运行“帮助(编解码器. codec)”以获取允许的编码错误字符串列表。

newline：控制通用换行符的工作方式(它只适用于文本模式)。它可以是None，’ ’，‘n’，‘r’，‘rn’。它的工作原理如下:

* 在输入时，如果换行为None，则启用通用换行模式。输入中的行可以以“n”、“r”或“rn”结尾，这些会在返回给调用方之前被转换为“n”。如果是’ ‘，则启用通用换行模式，但行尾将返回给调用方，未翻译。如果它有任何其他合法值，输入行仅由给定的字符串终止，并且行结束返回给调用方，未翻译。
* 在输出端，如果换行符为None，则写入的任何“n”字符都被转换为系统默认的行分隔符os.linesep。如果换行符是“或‘n’，则不进行翻译。如果换行符是任何其他合法值，则写入的任何“n”字符都被转换为给定字符串。

colsefd: 如果closefd为False，那么在关闭文件时，底层文件描述符将保持打开状态。当给定文件名时，此方法无效，在这种情况下必须为True。

read,readline,readlines区别

* **file.read([size])**：size 未指定则返回整个文件，如果文件大小 >2 倍内存则有问题，f.read()读到文件尾时返回""(空字串)。
* **file.readline()**：返回一行。
* **file.readlines([size])**：返回包含size行的列表, size 未指定则返回全部行。
* **for line in f: print line**：通过迭代器访问。
* **f.write("hello\n")**：如果要写入字符串以外的数据,先将他转换为字符串。
* **f.tell()**：返回一个整数,表示当前文件指针的位置(就是到文件头的比特数)。
* **f.seek(偏移量,[起始位置])**：用来移动文件指针。

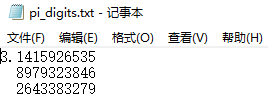
偏移量: 单位为比特，可正可负

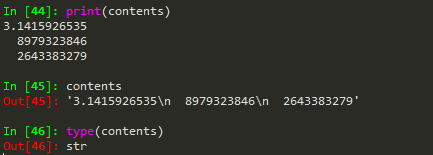
起始位置: 0 - 文件头, 默认值; 1 - 当前位置; 2 - 文件尾

* f.close()**关闭文件**

以一个简单文件为例：

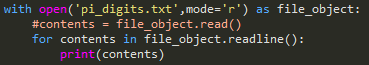
**.read() 每次读取整个文件，它通常将读取到底文件内容放到一个字符串变量中，也就是说 .read() 生成文件内容是一个字符串类型，如下图；**

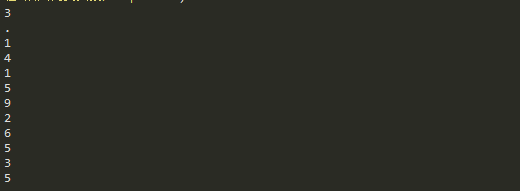




注意使不使用print的结果是不一样。

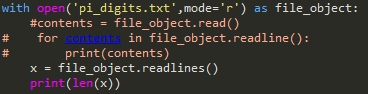
**.readline()每只读取文件的一行，通常也是读取到的一行内容放到一个字符串变量中，返回str类型，如下图；**





从for循环中我们可以看出.readline()只读取了文件中的一行内容。

**.readlines()每次按行读取整个文件内容，将读取到的内容放到一个列表中，返回list类型，如下图。**



**4. os模块用法**

OS模块是Python标准库中的一个用于访问操作系统功能的模块，OS模块提供了一种可移植的方法使用操作系统的功能。使用OS模块中提供的接口，可以实现跨平台访问。但是在OS模块中的接口并不是所有平台都通用，有些接口的实现是依靠特定平台下的接口的。在OS模块中提供了一系列访问操作系统功能的接口，便于编写跨平台的应用。

常见函数列表

os.access('pathfile',os.W\_OK) 检验文件权限模式，输出True，False

os.chmod('pathfile',os.W\_OK) 改变文件权限模式

os.chmod(file):修改文件权限和时间戳

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

参数 os.access(pathfile,mode)

* **pathfile** -- 要用来检测是否有访问权限的路径。
* **mode** -- mode为F\_OK，测试存在的路径，或者它可以是包含R\_OK, W\_OK和X\_OK或者R\_OK, W\_OK和X\_OK其中之一或者更多。
  + **os.F\_OK:** 测试path是否存在。是access()的mode默认参数
  + **os.R\_OK:** 测试path是否可读。
  + **os.W\_OK:** 测试path是否可写。
  + **os.X\_OK**：测试path是否可执行。

**返回值**

如果允许访问返回 True , 否则返回False。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# echo 'test' > test.sh

>>> os.access('test.sh',os.W\_OK)

True

>>> os.access('test.sh',os.X\_OK)

False

>>> os.chmod('test.sh',os.X\_OK)

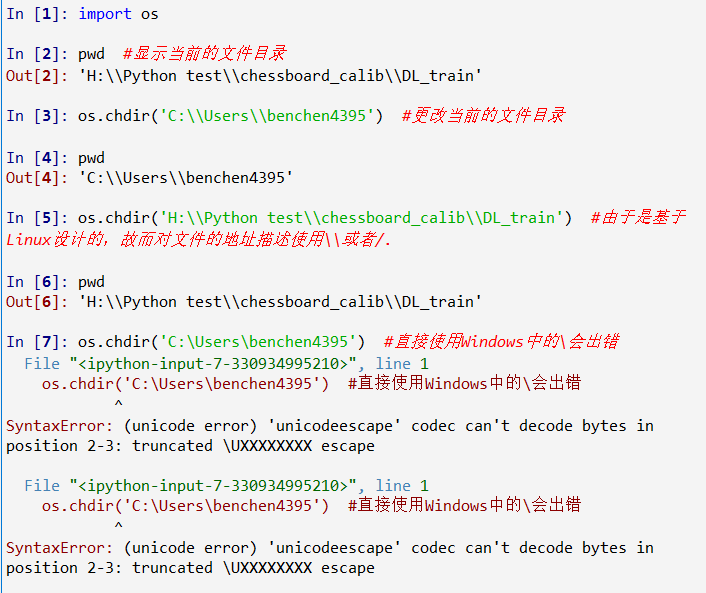
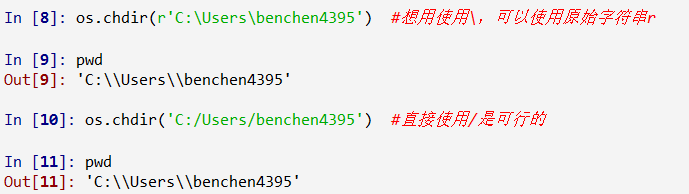
>>> os.access('test.sh',os.X\_OK)

True

# ls -l test.sh

---------x 1 root root 12 Oct 20 23:03 test.sh

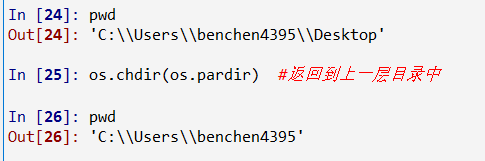
os.chdir(dirname):改变工作目录到dirname，相当于shell中的cd

os.curdir:返回当前目录的字符串名（'.'）；常用在程序中指代当前的目录地址，比如使用os.listdir(os.curdir)，就能返回os.curdir表示的目录中的所有文件。相当于在cmd执行dir；或在Linux中的ls

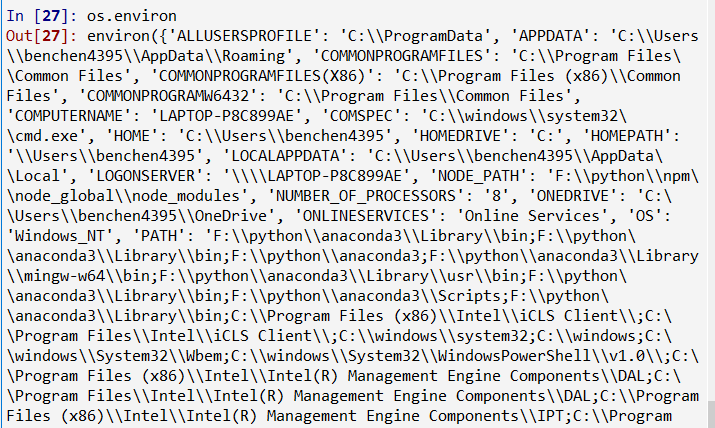


os.pardir 获取当前目录的父目录字符串名：('..')，相似与os.curdir，但返回上一层子目录



os.exit():终止当前进程

os.environ 获取系统环境变量



对于官方的解释，environ是一个字符串所对应环境的映像对象。这是什么意思呢？举个例子来说，environ['HOME']就代表了当前这个用户的主目录。



下图是windows和ubuntu下的environ的key列表：



作为一个渗透测试学习者来说，对系统的足够了解是基本的要求，下面就通过对os.environ中的key解读的角度来认识系统。

windows：

· os.environ['HOMEPATH']:当前用户主目录。

os.environ['TEMP']:临时目录路径。

os.environ[PATHEXT']:可执行文件。

os.environ['SYSTEMROOT']:系统主目录。

os.environ['LOGONSERVER']:机器名。

os.environ['PROMPT']:设置提示符。

linux：

os.environ['USER']:当前使用用户。

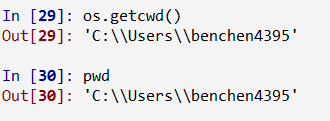
os.environ['LC\_COLLATE']:路径扩展的结果排序时的字母顺序。

os.environ['SHELL']:使用shell的类型。

os.environ['LAN']:使用的语言。

os.environ['SSH\_AUTH\_SOCK']:ssh的执行路径。

os.getcwd():获得当前工作目录，即当前python脚本工作的目录路径，相当于Windows中的cd 或Linux中的pwd



os.getenv()和os.putenv:分别用来读取和设置环境变量

原型：os.getenv (varname)

解释：返回当前进程的环境变量varname的值,若变量没有定义时返回nil

Usage

首先新建一个文件命名为getenvtest.lua然后编写如下代码：

print("start test getenv() function:\n")

print(os.getenv("SystemRoot")) -- 系统根目录

print(os.getenv("WoXiaXieDe")) -- 我乱写的

print(os.getenv("ALLUSERSPROFILE")) -- 所有“用户配置文件”的位置

print(os.getenv("alluserSpRoFilE")) -- 所有“用户配置文件”的位置

print(os.getenv("COMPUTERNAME")) -- 计算机的名称

print("\n")

print(os.getenv("COMSPEC")) -- 命令行解释器可执行程序的准确路径

print(os.getenv("HOMEDRIVE")) -- 连接到用户主目录的本地工作站驱动器号

print(os.getenv("HOMEPATH")) -- 用户主目录的完整路径

print(os.getenv("NUMBER\_OF\_PROCESSORS"))-- 安装在计算机上的处理器的数目

print(os.getenv("OS")) -- 操作系统的名称

print("\n")

print(os.getenv("PROCESSOR\_LEVEL")) -- 计算机上安装的处理器的型号

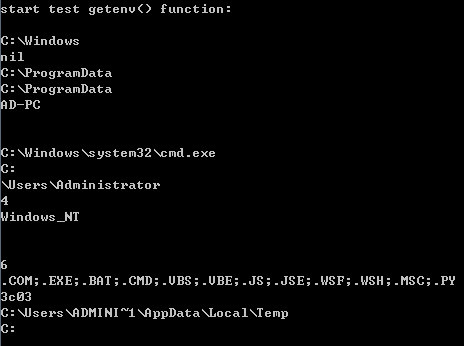
print(os.getenv("PATHEXT")) -- 连接到用户主目录的本地工作站驱动器号

print(os.getenv("PROCESSOR\_REVISION")) -- 处理器修订号的系统变量

print(os.getenv("TEMP")) -- 临时目录

print(os.getenv("SYSTEMDRIVE")) -- 系统根目录的驱动器

运行结果



getenv.png

总结

由结果可知，函数os.getenv(varname)对于参数中的varname是不区分大小写的。

当varname不存在时，函数会返回nil作为结果返回。

os.listdir(dirname):列出dirname下的目录和文件，包括隐藏文件，并以列表方式打印

os.linesep:给出当前平台的行终止符。例如，Windows使用'\r\n'，Linux使用'\n'而Mac使用'\r'



os.mkdir(name):创建目录

os.makedirs('dirname1/dirname2') 可生成多层递归目录

os.name:指示你正在使用的工作平台。比如对于Windows，它是'nt'，而对于Linux/Unix用户，它是'posix'。

os.pathsep 输出用于分割文件路径的字符串

os.pardir 获取当前目录的父目录字符串名：('..')

os.path.abspath(path) #返回绝对路径

>>> import os.path

>>> os.path.abspath('c.py')

'/root/py/c.py'

>>> os.path.abspath('../py/c.py')

'/root/py/c.py'

注：使用os.path.abspath(path)获取文件的绝对路径后，os.path.split(path)命令会将路径进行分割成（文件所在目录+文件名）两部分，os.path.basename(path)返回文件名，os.path.dirname(path)返回文件所在目录。

os.path.basename(path) #返回path最后的文件名。如何path以／或\结尾，那么就会返回空值。即os.path.split(path)的第二个元素

>>> os.path.basename('/root/py/c.py')

'c.py'

>>> os.path.basename('/root/py')

'py'

os.path.commonprefix(list) #返回list(多个路径)中，所有path共有的最长的路径。

os.path.dirname(path) #返回文件路径，其实就是os.path.split(path)的第一个元素

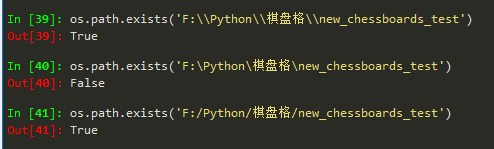
>>> os.path.dirname('/root/py/c.py')

'/root/py'

>>> os.path.dirname('c.py')

''

os.path.exists(path)  #路径存在则返回True,路径损坏返回False



os.path.lexists  #路径存在则返回True,路径损坏也返回True

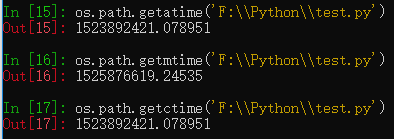
os.path.expanduser(path)  #把path中包含的"~"和"~user"转换成用户目录

os.path.expandvars(path)  #根据环境变量的值替换path中包含的”$name”和”${name}”

os.path.getatime(path)  #返回该路径对应文件的最近一次访问时间的时间戳（秒），如果文件不存在或无法访问，则引发OSError

os.path.getmtime(path)  #返回该路径对应文件的最后修改时间的时间戳（秒），如果文件不存在或无法访问，则引发OSError

os.path.getctime(path)  #返回该路径对应文件的ctime，在某些系统上（如Unix上）是最后一次元数据更改时间，在其他系统上（如Windows）是路径的创建时间；如果文件不存在或无法访问，则引发OSError



os.path.getsize(path)  #返回路径大小（字节），如果文件不存在就返回错误



os.path.isabs(path)  #判断是否为绝对路径

os.path.isfile(path)  #判断路径是否为文件

os.path.isdir(path)  #判断路径是否为目录

os.path.islink(path)  #判断路径是否为链接

os.path.ismount(path)  #判断路径是否为挂载点（）

os.path.join(path,name) #把目录和文件名合成一个路径，如果在拼接过程中遇到绝对路径将会丢弃前面的部分并从该绝对路径重新开始拼接

a = '\t\tab\nc'

是在拼接路径的时候用的。举个例子

os.path.join(“home”, "me", "mywork")

在Linux系统上会返回

“home/me/mywork"

在Windows系统上会返回

"home\me\mywork"

好处是可以根据系统自动选择正确的路径分隔符"/"或"\"

os.path.normcase(path)  #转换path的大小写和斜杠；在Linux下，该函数会原样返回path，在windows平台上会将路径中所有字符转换为小写，并将所有斜杠转换为反斜杠

>>> os.path.normcase('c:/windows\\system32\\')

'c:\\windows\\system32\\'

os.path.normpath(path)  #规范path字符串形式

>>> os.path.normpath('c://windows\\System32\\../Temp/')

'c:\\windows\\Temp'

os.path.realpath(path)  #返回path的真实路径

os.path.relpath(path[, start])  #从start开始计算相对路径

os.removedirs('dirname1') 若目录为空，则删除，并递归到上一级目录，如若也为空，则删除，依此类推

os.rmdir('dirname') 删除单级空目录，若目录不为空则无法删除，报错；相当于shell中rmdir dirname

os.path.samefile(path1, path2)  #判断目录或文件是否相同

os.path.sameopenfile(fp1, fp2)  #判断fp1和fp2是否指向同一文件

os.path.samestat(stat1, stat2)  #判断stat tuple stat1和stat2是否指向同一个文件

os.path.split(path)  #把路径分割成dirname和basename，返回一个元组。分割文件名与目录（事实上，如果你完全使用目录，它也会将最后一个目录作为文件名而分离，同时它不会判断文件或目录是否存在）

>>> os.path.split('/root/py/c.py')

('/root/py', 'c.py')

>>> os.path.split('/root/py/')

('/root/py', '')

os.path.splitdrive(path)   #一般用在windows下，返回驱动器名和路径组成的元组；对linux元组第一个总是空的

>>> os.path.splitdrive('c:\\windows')

('c:', '\\windows')

os.path.splitext(path)  #分割路径，返回路径名和文件扩展名的元组；默认返回(fname,fextension)元组，可做分片操作 ，以“.”为分隔符

>>> os.path.splitext('/root/py/c.py')

('/root/py/c', '.py')

os.path.splitunc(path)  #把路径分割为加载点与文件

os.path.walk(top,func,arg)

top表示需要遍历的目录树的路径

func表示回调函数，对遍历路径进行处理.所谓回调函数，是作为某个函数的参数使用，当某个时间触发时，程序将调用定义好的回调函数处理某个任务.回调函数必须提供3个参数：第1个参数为walk()的参数arg，第2个参数表示目录列表，第3个参数表示文件列表

arg是传递给回调参数func的元组.回调函数的一个参数必须是arg，为回调函数提供处理参数.参数arg可以为空

>>> import os

>>> def VisitDir(arg,dirname,names):

... for filespath in names:

...  print os.path.join(dirname,filespath)

...

>>> path='/root/py/wd/chat'

>>> os.path.walk(path,VisitDir,())

/root/py/wd/chat/chat\_server.py

/root/py/wd/chat/chat\_client.py

/root/py/wd/chat/test

/root/py/wd/chat/test/linuxeye

/root/py/wd/chat/test/test2

/root/py/wd/chat/test/test3

/root/py/wd/chat/test/test2/asdf

/root/py/wd/chat/test/test3/sdfaxx

os.path.walk()与os.walk()产生的文件名列表并不相同。os.path.walk()产生目录树下的目录路径和文件路径，而os.walk()只产生文件路径

os.path.supports\_unicode\_filenames  #设置是否支持unicode路径名

os.system("bash command") 运行shell命令，直接显示

os.popen("bash command") 运行shell命令，生成对象，可赋给变量，再用read读取

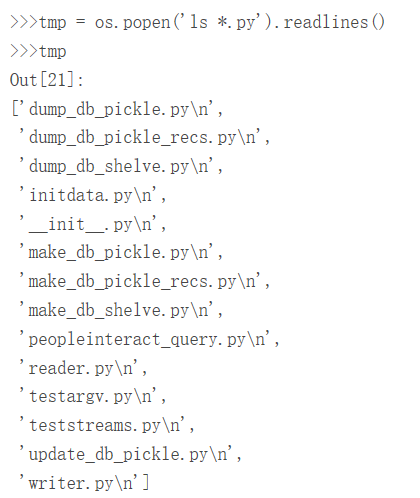
1. os.system() # 相当于在cmd中运行命令。仅仅在一个子终端运行系统命令,而不能获取命令执行后的返回信息. 即不能将返回信息赋予一个特定的值。

>>> os.system('ls')  #如果再命令行下执行,结果直接打印出来

1. 04101419778.CHM  bash     document   media     py-django  video
2. 11.wmv           books    downloads  Pictures  python
3. all-20061022     Desktop  Examples   project   tools

2os.popen() # 此种方法不但执行命令还返回执行后的信息对象

将返回的结果赋于一变量,便于程序的处理.



>>> import os

>>> os.system('ls twisted')

chat\_client\_twisted.py chat\_server\_twisted.py

0

>>> LS = os.popen('ls twisted')

>>> LS.readlines()

['chat\_client\_twisted.py\n', 'chat\_server\_twisted.py\n']

os.remove(file):删除一个文件

os.removedirs（r“c：\python”）:删除多个目录

os.rename("oldname","newname") 重命名文件/目录

os.rmdir(name):删除目录

os.sep:取代操作系统特定的路径分隔符；输出操作系统特定的路径分隔符，win下为"\\",Linux下为"/"

os.stat（file）:获得文件属性

os.system():运行shell命令

os.symlink('path/filename','ln\_filename') 创建符号链接，源需绝对路径

os.tmpfile() 创建并打开‘w+b'一个新的临时文件

os.utime() 修改时间属性

>>> import os

>>> stinfo = os.stat('c.py')

>>> print "access time of c.py: %s \nmodified time of c.py: %s" % (stinfo.st\_atime,stinfo.st\_mtime)

access time of c.py: 1375448908.0

modified time of c.py: 1369735909.0

>>> os.utime('c.py',(1375448978,1369735977))

>>> print "access time of c.py: %s \nmodified time of c.py: %s" % (stinfo.st\_atime,stinfo.st\_mtime)

access time of c.py: 1375448908.0

modified time of c.py: 1369735909.0

退出Python交互模式，再次进入

>>> import os

>>> stinfo = os.stat('c.py')

>>> print "access time of c.py: %s \nmodified time of c.py: %s" % (stinfo.st\_atime,stinfo.st\_mtime)

access time of c.py: 1375448978.0

modified time of c.py: 1369735977.0

os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])

1. top表示需要遍历的目录树的路径
2. topdown的默认值是”True”,表示首先返回目录树下的文件，然后在遍历目录树的子目录.Topdown的值为”False”时，则表示先遍历目录树的子目录，返回子目录下的文件，最后返回根目录下的文件
3. onerror的默认值是”None”,表示忽略文件遍历时产生的错误.如果不为空，则提供一个自定义函数提示错误信息后继续遍历或抛出异常中止遍历

该函数返回一个元组，该元组有3个元素，这3个元素分别表示每次遍历的路径名，目录列表和文件列表  
**os.walk()举例：**

>>> import os

>>> for root, dirs, files in os.walk("wd/chat", topdown=False):

... for name in files:

...  print(os.path.join(root, name)) #打印文件绝对路径

... for name in dirs:

...  print(os.path.join(root, name)) #打印目录绝对路径

...

文件操作（http://www.cnblogs.com/cherishry/p/5725977.html）

os.mknod("text.txt")：创建空文件

fp = open("text.txt",w):直接打开一个文件，如果文件不存在就创建文件

关于open的模式

w 写方式

a 追加模式打开（从EOF开始，必要时创建新文件）

r+ 以读写模式打开

w+ 以读写模式打开

a+ 以读写模式打开

rb 以二进制读模式打开

wb 以二进制写模式打开 (参见 w )

ab 以二进制追加模式打开 (参见 a )

rb+ 以二进制读写模式打开 (参见 r+ )

wb+ 以二进制读写模式打开 (参见 w+ )

ab+ 以二进制读写模式打开 (参见 a+ )

关于文件的函数

fp.read([size])

size为读取的长度，以byte为单位

fp.readline([size])

读一行，如果定义了size，有可能返回的只是一行的一部分

fp.readlines([size])

把文件每一行作为一个list的一个成员，并返回这个list。其实它的内部是通过循环调用readline()来实现的。如果提供size参数，size是表示读取内容的总长，也就是说可能只读到文件的一部分。

fp.write(str)

把str写到文件中，write()并不会在str后加上一个换行符

fp.writelines(seq)

把seq的内容全部写到文件中(多行一次性写入)。这个函数也只是忠实地写入，不会在每行后面加上任何东西。

fp.close()

关闭文件。python会在一个文件不用后自动关闭文件，不过这一功能没有保证，最好还是养成自己关闭的习惯。 如果一个文件在关闭后还对其进行操作会产生ValueError

fp.flush()

把缓冲区的内容写入硬盘

fp.fileno()

返回一个长整型的”文件标签“

fp.isatty()

文件是否是一个终端设备文件（unix系统中的）

fp.tell()

返回文件操作标记的当前位置，以文件的开头为原点

fp.next()

返回下一行，并将文件操作标记位移到下一行。把一个file用于for … in file这样的语句时，就是调用next()函数来实现遍历的。

fp.seek(offset[,whence])

将文件打操作标记移到offset的位置。这个offset一般是相对于文件的开头来计算的，一般为正数。但如果提供了whence参数就不一定了，whence可以为0表示从头开始计算，1表示以当前位置为原点计算。2表示以文件末尾为原点进行计算。需要注意，如果文件以a或a+的模式打开，每次进行写操作时，文件操作标记会自动返回到文件末尾。

fp.truncate([size])

把文件裁成规定的大小，默认的是裁到当前文件操作标记的位置。如果size比文件的大小还要大，依据系统的不同可能是不改变文件，也可能是用0把文件补到相应的大小，也可能是以一些随机的内容加上去。

目录操作

os.mkdir("file")

创建目录

复制文件:

shutil.copyfile("oldfile","newfile")

oldfile和newfile都只能是文件

shutil.copy("oldfile","newfile")

oldfile只能是文件夹，newfile可以是文件，也可以是目标目录

shutil.copytree("olddir","newdir")

复制文件夹.olddir和newdir都只能是目录，且newdir必须不存在

os.rename("oldname","newname")

重命名文件（目录）.文件或目录都是使用这条命令

shutil.move("oldpos","newpos")

移动文件（目录）

os.rmdir("dir")

只能删除空目录

shutil.rmtree("dir")

空目录、有内容的目录都可以删

os.chdir("path")

转换目录，换路径

代码演示

#!/usr/bin/env python

import os

print os.getcwd() #获取当前路径

print os.listdir('/mnt') #列出/mnt下面的所有目录和文件

print os.mkdir('lala') #创建目录lala和haha

print os.mkdir('haha')

print os.rmdir('haha') #删除haha

print os.rename('example.txt','back.txt') #修改文件名

运行：

root@ruanyang-HP-ProDesk-680-G2-MT:/mnt/python# python os.py

/mnt/python

['proc', 'perl\_bash', 'docker', 'warn', 'blog', 'python', 'hostname', 'test', 'tags', 'perl\_DB', 'shell', 'perl\_test', '.vimrc~', 'club.yml', 'test.c', '.vimrc']

None

None

None

None

root@ruanyang-HP-ProDesk-680-G2-MT:/mnt/python# ls

back.txt exit.py lala modules.py os\_file.py os.py sys.py

**5. 正则表达式（re模块）**

**5.1、参考文献**

1、https://www.cnblogs.com/tina-python/p/5508402.html

2、Python网络爬虫实战（胡松涛 编著） P102

3、Python基础教程（第二版-修订版） P191

**5.2、正则表达式中常用的字符含义**

来自 <[*https://www.cnblogs.com/tina-python/p/5508402.html*](https://www.cnblogs.com/tina-python/p/5508402.html)>

**1、普通字符和11个元字符：**

|  |  |  |  |
| --- | --- | --- | --- |
| 普通字符 | 匹配自身 | abc | abc |
| . | 匹配任意除换行符"\n"外的字符(在DOTALL模式中也能匹配换行符） | a.c | abc |
| \ | 转义字符，使后一个字符改变原来的意思 | a\.c;a\\c | a.c;a\c |
| \* | 匹配前一个字符0或多次 | abc\* | ab;abccc |
| + | 匹配前一个字符1次或无限次 | abc+ | abc;abccc |
| ? | 匹配一个字符0次或1次 | abc? | ab;abc |
| ^ | 匹配字符串开头。在多行模式中匹配每一行的开头，如^abc匹配以abc为开头的字符串 | ^abc | abc |
| $ | 匹配字符串末尾，在多行模式中匹配每一行的末尾 | abc$ | abc |
| | | 或。匹配|左右表达式任意一个，从左到右匹配，如果|没有包括在()中，则它的范围是整个正则表达式 | abc|def | abc  def |
| {} | {m}匹配前一个字符m次，{m,n}匹配前一个字符m至n次，若省略n，则匹配m至无限次 | ab{1,2}c | abc  abbc |
| [] | 字符集。对应的位置可以是字符集中任意字符。字符集中的字符可以逐个列出，也可以给出范围，如[abc]或[a-c]。[^abc]表示取反，即非abc。  所有特殊字符在字符集中都失去其原有的特殊含义。用\反斜杠转义恢复特殊字符的特殊含义。 | a[bcd]e | abe  ace  ade |
| () | 被括起来的表达式将作为分组，从表达式左边开始每遇到一个分组的左括号“（”，编号+1.  分组表达式作为一个整体，可以后接数量词。表达式中的|仅在该组中有效。 | (abc){2}  a(123|456)c | abcabc  a456c |

这里需要强调一下反斜杠\的作用：

* 反斜杠后边跟元字符去除特殊功能；（即将特殊字符转义成普通字符）
* 反斜杠后边跟普通字符实现特殊功能；（即预定义字符）
* 引用序号对应的字组所匹配的字符串。

a=re.search(r'(tina)(fei)haha\2','tinafeihahafei tinafeihahatina').group()  
print(a)  
结果：  
Tinafeihahafei

**2、预定义字符集（可以写在字符集[...]中）**

|  |  |  |  |
| --- | --- | --- | --- |
| \d | 数字:[0-9] | a\bc | a1c |
| \D | 非数字:[^\d] | a\Dc | abc |
| \s | 匹配任何空白字符:[<空格>\t\r\n\f\v] | a\sc | a c |
| \S | 非空白字符:[^\s] | a\Sc | abc |
| \w | 匹配包括下划线在内的任何字字符:[A-Za-z0-9\_] | a\wc | abc |
| \W | 匹配非字母字符，即匹配特殊字符 | a\Wc | a c |
| \A | 仅匹配字符串开头,同^ | \Aabc | abc |
| \Z | 仅匹配字符串结尾，同$ | abc\Z | abc |
| \b | 匹配\w和\W之间，即匹配单词边界匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。 | \babc\b  a\b!bc | 空格abc空格  a!bc |
| \B | [^\b] | a\Bbc | abc |

这里需要强调一下\b的单词边界的理解：

w = re.findall('\btina','tian tinaaaa')

print(w)

s = re.findall(r'\btina','tian tinaaaa')

print(s)

v = re.findall(r'\btina','tian#tinaaaa')

print(v)

a = re.findall(r'\btina\b','tian#tina@aaa')

print(a)

执行结果如下：

[]

['tina']

['tina']

['tina']

**3、特殊分组用法：**

|  |  |  |  |
| --- | --- | --- | --- |
| (?P<name>) | 分组，除了原有的编号外再指定一个额外的别名 | (?P<id>abc){2} | abcabc |
| (?P=name) | 引用别名为<name>的分组匹配到字符串 | (?P<id>\d)abc(?P=id) | 1abc1  5abc5 |
| \<number> | 引用编号为<number>的分组匹配到字符串 | (\d)abc\1 | 1abc1  5abc5 |

**5.3、re模块中常用功能函数**

**1、compile()**

编译正则表达式模式，返回一个对象的模式。（可以把那些常用的正则表达式编译成正则表达式对象，这样可以提高一点效率。）

格式：

re.compile(pattern,flags=0)

pattern: 编译时用的表达式字符串。

flags 编译标志位，用于修改正则表达式的匹配方式，如：是否区分大小写，多行匹配等。常用的flags有：

|  |  |
| --- | --- |
| 标志 | 含义 |
| re.S(DOTALL) | 使.匹配包括换行在内的所有字符 |
| re.I（IGNORECASE） | 使匹配对大小写不敏感 |
| re.L（LOCALE） | 做本地化识别（locale-aware)匹配，法语等 |
| re.M(MULTILINE) | 多行匹配，影响^和$ |
| re.X(VERBOSE) | 该标志通过给予更灵活的格式以便将正则表达式写得更易于理解 |
| re.U | 根据Unicode字符集解析字符，这个标志影响\w,\W,\b,\B |

import re  
tt = "Tina is a good girl, she is cool, clever, and so on..."  
rr = re.compile(r'\w\*oo\w\*')  
print(rr.findall(tt)) #查找所有包含'oo'的单词  
执行结果如下：  
['good', 'cool']

**2、match()**

决定RE是否在字符串刚开始的位置匹配。//注：这个方法并不是完全匹配。当pattern结束时若string还有剩余字符，仍然视为成功。想要完全匹配，可以在表达式末尾加上边界匹配符'$'

格式：

re.match(pattern, string, flags=0)

print(re.match('com','comwww.runcomoob').group())  
print(re.match('com','Comwww.runcomoob',re.I).group())  
执行结果如下：  
com  
com

**3、search()**

 格式：

re.search(pattern, string, flags=0)

re.search函数会在字符串内查找模式匹配,只要找到第一个匹配然后返回，如果字符串没有匹配，则返回None。

print(re.search('\dcom','www.4comrunoob.5com').group())  
执行结果如下：  
4com

\*注：match和search一旦匹配成功，就是一个match object对象，而match object对象有以下方法：

* group() 返回被 RE 匹配的字符串
* start() 返回匹配开始的位置
* end() 返回匹配结束的位置
* span() 返回一个元组包含匹配 (开始,结束) 的位置
* group() 返回re整体匹配的字符串，可以一次输入多个组号，对应组号匹配的字符串。

a. group（）返回re整体匹配的字符串，

b. group (n,m) 返回组号为n，m所匹配的字符串，如果组号不存在，则返回indexError异常

c.groups（）groups() 方法返回一个包含正则表达式中所有小组字符串的元组，从 1 到所含的小组号，通常groups()不需要参数，返回一个元组，元组中的元就是正则表达式中定义的组。

import re  
a = "123abc456"  
 print(re.search("([0-9]\*)([a-z]\*)([0-9]\*)",a).group(0)) #123abc456,返回整体  
 print(re.search("([0-9]\*)([a-z]\*)([0-9]\*)",a).group(1)) #123  
 print(re.search("([0-9]\*)([a-z]\*)([0-9]\*)",a).group(2)) #abc  
 print(re.search("([0-9]\*)([a-z]\*)([0-9]\*)",a).group(3)) #456

###group(1) 列出第一个括号匹配部分，group(2) 列出第二个括号匹配部分，group(3) 列出第三个括号匹配部分。###

**4、findall()**

re.findall遍历匹配，可以获取字符串中所有匹配的字符串，返回一个列表。

 格式：

re.findall(pattern, string, flags=0)

p = re.compile(r'\d+')  
print(p.findall('o1n2m3k4'))  
执行结果如下：  
['1', '2', '3', '4']

import re  
tt = "Tina is a good girl, she is cool, clever, and so on..."  
rr = re.compile(r'\w\*oo\w\*')  
print(rr.findall(tt))  
print(re.findall(r'(\w)\*oo(\w)',tt))#()表示子表达式   
执行结果如下：  
['good', 'cool']  
[('g', 'd'), ('c', 'l')]

**5、finditer()**

 搜索string，返回一个顺序访问每一个匹配结果（Match对象）的迭代器。找到 RE 匹配的所有子串，并把它们作为一个迭代器返回。

格式：

re.finditer(pattern, string, flags=0)

iter = re.finditer(r'\d+','12 drumm44ers drumming, 11 ... 10 ...')  
for i in iter:  
 print(i)  
 print(i.group())  
 print(i.span())  
执行结果如下：  
<\_sre.SRE\_Match object; span=(0, 2), match='12'>  
12  
(0, 2)  
<\_sre.SRE\_Match object; span=(8, 10), match='44'>  
44  
(8, 10)  
<\_sre.SRE\_Match object; span=(24, 26), match='11'>  
11  
(24, 26)  
<\_sre.SRE\_Match object; span=(31, 33), match='10'>  
10  
(31, 33)

**6、split()**

按照能够匹配的子串将string分割后返回列表。

可以使用re.split来分割字符串，如：re.split(r'\s+', text)；将字符串按空格分割成一个单词列表。

格式：

re.split(pattern, string[, maxsplit])

maxsplit用于指定最大分割次数，不指定将全部分割。

print(re.split('\d+','one1two2three3four4five5'))  
执行结果如下：  
['one', 'two', 'three', 'four', 'five', '']

**7、sub()**

使用re替换string中每一个匹配的子串后返回替换后的字符串。

格式：

re.sub(pattern, repl, string, count)

import re  
text = "JGood is a handsome boy, he is cool, clever, and so on..."  
print(re.sub(r'\s+', '-', text))  
执行结果如下：  
JGood-is-a-handsome-boy,-he-is-cool,-clever,-and-so-on...

其中第二个函数是替换后的字符串；本例中为'-'

第四个参数指替换个数。默认为0，表示每个匹配项都替换。

re.sub还允许使用函数对匹配项的替换进行复杂的处理。

如：re.sub(r'\s', lambda m: '[' + m.group(0) + ']', text, 0)；将字符串中的空格' '替换为'[ ]'。

import re  
text = "JGood is a handsome boy, he is cool, clever, and so on..."  
print(re.sub(r'\s+', lambda m:'['+m.group(0)+']', text,0))  
执行结果如下：  
JGood[ ]is[ ]a[ ]handsome[ ]boy,[ ]he[ ]is[ ]cool,[ ]clever,[ ]and[ ]so[ ]on...

**8、subn()**

 返回替换次数

格式：

subn(pattern, repl, string, count=0, flags=0)

print(re.subn('[1-2]','A','123456abcdef'))  
print(re.sub("g.t","have",'I get A, I got B ,I gut C'))  
print(re.subn("g.t","have",'I get A, I got B ,I gut C'))  
执行结果如下：  
('AA3456abcdef', 2)  
I have A, I have B ,I have C  
('I have A, I have B ,I have C', 3)

**5.4、一些注意点**

**1、re.match与re.search与re.findall的区别：**

re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；而re.search匹配整个字符串，直到找到一个匹配。

[复制代码](javascript:void(0);)

a=re.search('[\d]',"abc33").group()  
print(a)  
p=re.match('[\d]',"abc33")  
print(p)  
b=re.findall('[\d]',"abc33")  
print(b)  
执行结果：  
3  
None  
['3', '3']

**2、贪婪匹配与非贪婪匹配**

\*?,+?,??,{m,n}?    前面的\*,+,?等都是贪婪匹配，也就是尽可能匹配，后面加?号使其变成惰性匹配

a = re.findall(r"a(\d+?)",'a23b')  
print(a)  
b = re.findall(r"a(\d+)",'a23b')  
print(b)  
执行结果：  
['2']  
['23']

a = re.match('<(.\*)>','<H1>title<H1>').group()  
print(a)  
b = re.match('<(.\*?)>','<H1>title<H1>').group()  
print(b)  
执行结果：  
<H1>title<H1>  
<H1>

a = re.findall(r"a(\d+)b",'a3333b')  
print(a)  
b = re.findall(r"a(\d+?)b",'a3333b')  
print(b)  
执行结果如下：  
['3333']  
['3333']  
#######################  
这里需要注意的是如果前后均有限定条件的时候，就不存在什么贪婪模式了，非匹配模式失效。

**3、用flags时遇到的小坑**

print(re.split('a','1A1a2A3',re.I))#输出结果并未能区分大小写  
这是因为re.split(pattern，string，maxsplit,flags)默认是四个参数，当我们传入的三个参数的时候，系统会默认re.I是第三个参数，所以就没起作用。如果想让这里的re.I起作用，写成flags=re.I即可。

**5.5、正则的小实践**

1、匹配电话号码

p = re.compile(r'\d{3}-\d{6}')  
print(p.findall('010-628888'))

2、匹配IP

re.search(r"(([01]?\d?\d|2[0-4]\d|25[0-5])\.){3}([01]?\d?\d|2[0-4]\d|25[0-5]\.)","192.168.1.1")

来自 <[*https://www.cnblogs.com/tina-python/p/5508402.html*](https://www.cnblogs.com/tina-python/p/5508402.html)>

**6. sklearn模块**

**6.1 sklearn.preprocessing 模块**

参考文献见：<http://sklearn.apachecn.org/cn/0.19.0/modules/preprocessing.html#preprocessing>

6.1.1 标准化，去均值和方差按比例缩放

数据集的 **标准化** 对scikit-learn中实现的大多数机器学习算法来说是 **常见的要求** 。如果个别特征或多或少看起来不是很像标准正态分布(**具有零均值和单位方差**)，那么它们的表现力可能会较差。

在实际情况中,我们经常忽略特征的分布形状，直接经过去均值来对某个特征进行中心化，再通过除以非常量特征(non-constant features)的标准差进行缩放。

例如，在机器学习算法的目标函数(例如SVM的RBF内核或线性模型的l1和l2正则化)，许多学习算法中目标函数的基础都是假设所有的特征都是零均值并且具有同一阶数上的方差。如果某个特征的方差比其他特征大几个数量级，那么它就会在学习算法中占据主导位置，导致学习器并不能像我们说期望的那样，从其他特征中学习。

1. scale()函数

函数 [**scale**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.scale.html#sklearn.preprocessing.scale) 为数组形状的数据集的标准化提供了一个快捷实现:

prep.scale(X, axis=0, with\_mean=True, with\_std=True, copy=True)

X : {array-like, sparse matrix}. The data to center and scale.

axis:=0表示按列对数据进行0均值单位方差化；=1表示按行对数据进行0均值和单位方差化

copy:为True时表示转化后的是一个新的数组，为False时表示在原数组上直接转换

>>> from sklearn import preprocessing

>>> import numpy as np

>>> X\_train = np.array([[ 1., -1., 2.],

... [ 2., 0., 0.],

... [ 0., 1., -1.]])

>>> X\_scaled = preprocessing.scale(X\_train，axis=0) #按列进行标准化

# 标准化后的X\_scaled的值

>>> X\_scaled

array([[ 0. ..., -1.22..., 1.33...],

[ 1.22..., 0. ..., -0.26...],

[-1.22..., 1.22..., -1.06...]])

经过缩放后的数据具有零均值以及标准方差:

>>> X\_scaled.mean(axis=0)

array([ 0., 0., 0.])

>>> X\_scaled.std(axis=0)

array([ 1., 1., 1.])

1. StandardScaler()类

prep.StandardScaler(copy=True, with\_mean=True, with\_std=True)

实现了转化器的API来计算训练集上的平均值和标准偏差，以便以后能够在测试集上重新应用相同的变换。

>>> X\_train = np.array([[ 1., -1., 2.],

... [ 2., 0., 0.],

... [ 0., 1., -1.]])

>>> scaler = preprocessing.StandardScaler().fit(X\_train)

>>> scaler

StandardScaler(copy=True, with\_mean=True, with\_std=True)

>>> scaler.mean\_

array([ 1. ..., 0. ..., 0.33...])

>>> scaler.scale\_ # 显示在训练数据集上计算的标准差

array([ 0.81..., 0.81..., 1.24...])

# 在训练集上执行标准化计算

>>> scaler.transform(X\_train)

array([[ 0. ..., -1.22..., 1.33...],

[ 1.22..., 0. ..., -0.26...],

[-1.22..., 1.22..., -1.06...]])

缩放类对象可以在新的数据上实现和训练集相同缩放操作:（用训练集的均值和方差进行计算）

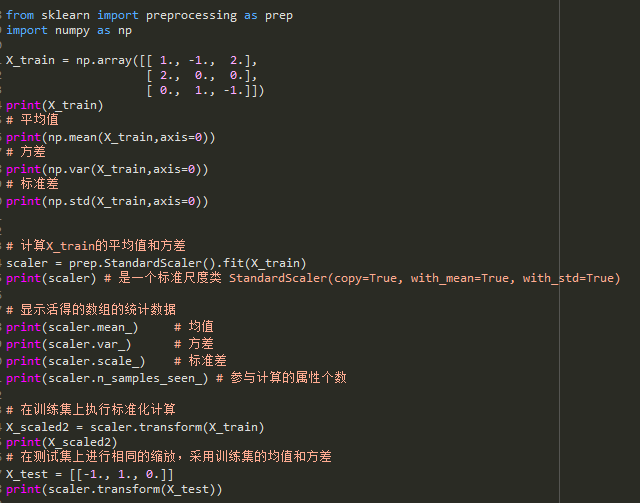
>>> X\_test = [[-1., 1., 0.]]

>>> scaler.transform(X\_test)

array([[-2.44..., 1.22..., -0.26...]])

你也可以通过在构造函数 :class:StandardScaler 中传入参数 with\_mean=False` 或者``with\_std=False 来取消中心化或缩放操作。

完整代码：



1. MinMaxScaler()类

一种标准化是将特征缩放到给定的最小值和最大值之间，通常在零和一之间，或者也可以将每个特征的最大绝对值转换至单位大小。可以分别使用 MinMaxScaler 和 MaxAbsScaler 实现。

使用这种缩放的目的包括实现特征极小方差的鲁棒性以及在稀疏矩阵中保留零元素。

以下是一个将简单的数据矩阵缩放到``[0, 1]``的例子:

>>> X\_train = np.array([[ 1., -1., 2.],

... [ 2., 0., 0.],

... [ 0., 1., -1.]])

...

>>> min\_max\_scaler = preprocessing.MinMaxScaler()

>>> X\_train\_minmax = min\_max\_scaler.fit\_transform(X\_train)

>>> X\_train\_minmax

array([[ 0.5 , 0. , 1. ],

[ 1. , 0.5 , 0.33333333],

[ 0. , 1. , 0. ]])

同样的转换实例可以被用与在训练过程中不可见的测试数据:实现和训练数据一致的缩放和移位操作:

>>> X\_test = np.array([[ -3., -1., 4.]])

>>> X\_test\_minmax = min\_max\_scaler.transform(X\_test)

>>> X\_test\_minmax

array([[-1.5 , 0. , 1.66666667]])

可以检查缩放器（scaler）属性，来观察在训练集中学习到的转换操作的基本性质:

>>> min\_max\_scaler.scale\_

array([ 0.5 , 0.5 , 0.33...])

>>> min\_max\_scaler.min\_

array([ 0. , 0.5 , 0.33...])

如果给 [**MinMaxScaler**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler) 提供一个明确的 feature\_range=(min, max) ，完整的公式是:

X\_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))

X\_scaled = X\_std \* (max - min) + min

完整代码是：



1. MaxAbsScaler()类

类 [MaxAbsScaler](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.MaxAbsScaler.html#sklearn.preprocessing.MaxAbsScaler) 的工作原理非常相似，但是它只通过除以每个特征的最大值将训练数据特征缩放至 [-1, 1] 范围内，这就意味着，训练数据应该是已经零中心化或者是稀疏数据。 例子::用先前例子的数据实现最大绝对值缩放操作。

以下是使用上例中数据运用这个缩放器的例子:

>>> X\_train = np.array([[ 1., -1., 2.],

... [ 2., 0., 0.],

... [ 0., 1., -1.]])

...

>>> max\_abs\_scaler = preprocessing.MaxAbsScaler()

>>> X\_train\_maxabs = max\_abs\_scaler.fit\_transform(X\_train)

>>> X\_train\_maxabs # doctest +NORMALIZE\_WHITESPACE^

array([[ 0.5, -1. , 1. ],

[ 1. , 0. , 0. ],

[ 0. , 1. , -0.5]])

>>> X\_test = np.array([[ -3., -1., 4.]])

>>> X\_test\_maxabs = max\_abs\_scaler.transform(X\_test)

>>> X\_test\_maxabs

array([[-1.5, -1. , 2. ]])

>>> max\_abs\_scaler.scale\_

array([ 2., 1., 2.])

在 [**scale**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.scale.html#sklearn.preprocessing.scale) 模块中进一步提供了方便的功能。当你不想创建对象时，可以使用如 [**minmax\_scale**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.minmax_scale.html#sklearn.preprocessing.minmax_scale) 以及 [**maxabs\_scale**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.maxabs_scale.html#sklearn.preprocessing.maxabs_scale) 。

1. 缩放稀疏（矩阵）数据

中心化稀疏(矩阵)数据会破坏数据的稀疏结构，因此很少有一个比较明智的实现方式。但是缩放稀疏输入是有意义的，尤其是当几个特征在不同的量级范围时。

|  |
| --- |
| [**MaxAbsScaler**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.MaxAbsScaler.html#sklearn.preprocessing.MaxAbsScaler) 以及 [**maxabs\_scale**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.maxabs_scale.html#sklearn.preprocessing.maxabs_scale) 是专为缩放数据而设计的，并且是缩放数据的推荐方法。但是， [**scale**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.scale.html#sklearn.preprocessing.scale) 和 [**StandardScaler**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler) 也能够接受 scipy.sparse 作为输入，只要参数 with\_mean=False 被准确传入它的构造器。否则会出现 ValueError 的错误，因为默认的中心化会破坏稀疏性，并且经常会因为分配过多的内存而使执行崩溃。 [**RobustScaler**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.RobustScaler.html#sklearn.preprocessing.RobustScaler) 不能适应稀疏输入，但你可以在稀疏输入使用 transform 方法。  注意，缩放器同时接受压缩的稀疏行和稀疏列(参见 scipy.sparse.csr\_matrix 以及 scipy.sparse.csc\_matrix )。任何其他稀疏输入将会 **转化为压缩稀疏行表示** 。为了避免不必要的内存复制，建议在上游(早期)选择CSR或CSC表示。  最后，最后，如果已经中心化的数据并不是很大，使用 toarray 方法将输入的稀疏矩阵显式转换为数组是另一种选择。 |

对稀疏数据的标准化是一个很大的研究方向，这一块需要详细的资料（后续完成）

1. 缩放有离群值的数据

如果你的数据包含许多异常值，使用均值和方差缩放可能并不是一个很好的选择。这种情况下，你可以使用 [**robust\_scale**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.robust_scale.html#sklearn.preprocessing.robust_scale) 以及 [**RobustScaler**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.RobustScaler.html#sklearn.preprocessing.RobustScaler) 作为替代品。它们对你的数据的中心和范围使用更有鲁棒性的估计。

参考:

更多关于中心化和缩放数据的重要性讨论在此FAQ中提及: [Should I normalize/standardize/rescale the data?](http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-16.html)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Scaling vs Whitening：有时候独立地中心化和缩放数据是不够的，因为下游的机器学习模型能够对特征之间的线性依赖做出一些假设(这对模型的学习过程来说是不利的)。

要解决这个问题，你可以使用 [sklearn.decomposition.PCA](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.decomposition.PCA.html#sklearn.decomposition.PCA) 或 [sklearn.decomposition.RandomizedPCA](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.decomposition.RandomizedPCA.html#sklearn.decomposition.RandomizedPCA) 并指定参数 whiten=True 来更多移除特征间的线性关联。

在回归中缩放目标变量

[scale](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.scale.html#sklearn.preprocessing.scale) 以及 [StandardScaler](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler) 可以直接处理一维数组。在回归中，缩放目标/相应变量时非常有用。

1. 核矩阵的中心化

如果你有一个核矩阵 K，它计算由函数phi定义的特征空间的点积，那么一个 [**KernelCenterer**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.preprocessing.KernelCenterer.html#sklearn.preprocessing.KernelCenterer) 类能够转化这个核矩阵，通过移除特征空间的平均值，使它包含由函数phi定义的内部产物。

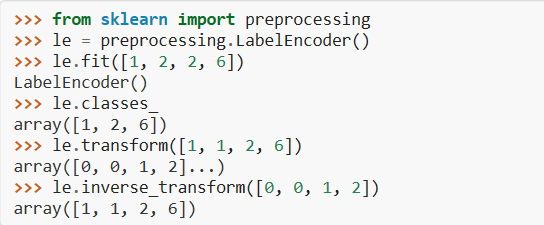
6.1.2 LabelEncoder(标签编码)

标准化标签值，即对标签的种类进行编码，取值从0至n-1；其中n为标签的种类。

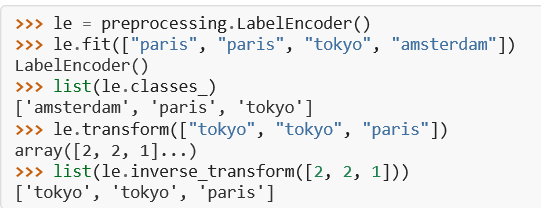
类中所包含的函数的用法：

|  |  |
| --- | --- |
| [**fit**](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.fit)(y) | 调整标签的编码，输入值y为一个(n,)的数组，返回值为LabelEncoder()。 |
| [**fit\_transform**](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.fit_transform)(y) | 调整标签的编码，并返回编码后的数字标签。输入的为[n]的array数组(但是例子中好像是list),输出为一个[n]的array数组，但是可以list化。 |
| [**get\_params**](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.get_params)([deep]) | Get parameters for this estimator. |
| [**inverse\_transform**](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.inverse_transform)(y) | 将已编码的标签转化为原始的标签类型。输入的为[n]的array数组(但是例子中好像是list),输出为一个[n]的array数组，但是可以list化。 |
| [**set\_params**](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.set_params)(\*\*params) | Set the parameters of this estimator. |
| [**transform**](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder.transform)(y) | 将标签转化为标准化的数字遍码。输入的为[n]的array数组(但是例子中好像是list),输出为一个[n]的array数组，但是可以list化。 |

1. 可以使原有的数字标签标准化：



1. 可以将原有的非数值标签(比如字符串标签)转化为数字标签（只要这些数据是hashable而且是可比较的）



注意：这里的list是将生成的数组(np.array)转化成list列表，可以不使用

（*http://scikitlearn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html*）

**7. pandas模块**

pandas含有使数据分析工作变得更快更简单的高级数据结构和操作工具。它是基于NumPy构建的，让以NumPy为中心的应用变得更加简单。

pandas基本功能：重新索引，丢弃指定轴上的项，索引、选取和过滤，算术运算和数据对齐，函数应用和映射，排序和排名，带有重复值的轴索引。

pandas可以以各种格式（包括数据库）输入输出数据、执行join以及其他SQL类似的功能来重塑数据、熟练地处理缺失值、支持时间序列、拥有基本绘图功能和统计功能，等等还有很多。

**7.1 pandas数据结构**

pandas的数据结构：Series、DataFrame、索引对象

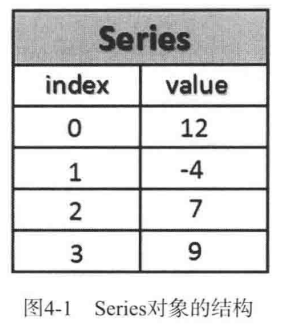
7.1.1 pandas常量

pandas空值的表示(None, np.NaN, np.NaT, pd.NaT)

* None: pandas对待None与np.nan类似。然而在很多情况下，the Python None 都会出现，and we wish to also consider that “missing” or “null”.
* NaN:不是一个数字，NaN是一个考虑计算的速度和方便性等种种原因而默认的缺失值标记( the default missing value marker)。我们需要能够轻松地用不同类型的数据来检测这个缺失值:浮点数、整数、布尔值和一般对象。
* NaT:Datetimes,对于datetime64 ns类型，NaT表示缺失的值。这是一个伪本地(pseudo-native)的标记值，可以用numpy的奇异值类型singular dtype(datetime64 ns)来表示。pandas对象提供了一个NaT and NaN之间的内部兼容
* inf: Prior to version v0.10.0 inf and -inf were also considered to be “null” in computations. This is no longer the case by default; use the mode.use\_inf\_as\_null option to recover it.

Note: 缺失值的判断要用np.isnan(),而不能使用a[0] == np.NaN.

7.1.2 Series

Series是一个一维的类似的数组对象，包含一个数组的数据（任何NumPy的数据类型）和一个与数组关联的数据标签，被叫做索引。

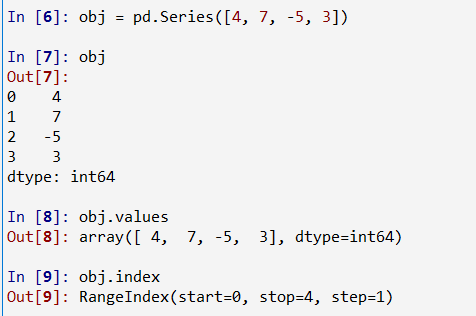
Seriers的交互式显示的字符串表示形式是索引在左边，值在右边。

总结说就是，它像一个数组，你可以像数组那样索引，它也像一个字典，你可以像字典那样索引。

**(1)series对象创建**

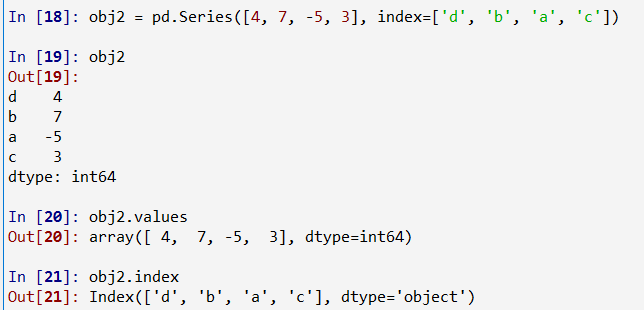
如果不给数据指定索引，一个包含整数0到 N-1 （这里N是数据的长度）的默认索引被创建。 你可以分别的通过它的values 和index 属性来获取Series的数组表示和索引对象：

最简单的Series是由一个数组的数据构成：



也就是它的索引是：Int64Index([0, 1, 2, 3])

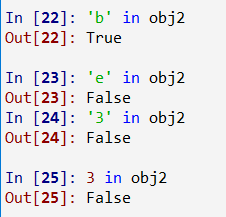
通常，需要创建一个带有索引来确定每一个数据点的Series：



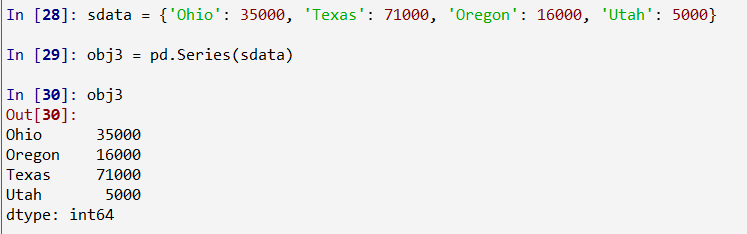
Note: pandas对象（series和dataframe）的index是可以修改的

Obj2.index = range(len(obj2))重新将index的值修改成了从0开始。这也许是和dict的一个不同吧。

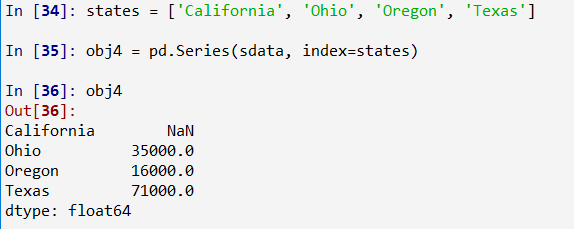
另一种思考的方式是，**Series是一个定长的，有序的字典**，因为它把索引和值映射起来了。它可以适用于许多需要一个字典的函数：



如果你有一些数据在一个Python字典中，你可以通过**传递字典来从这些数据创建一个Series**：

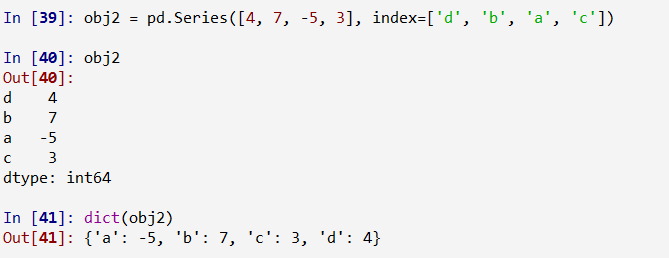


只传递一个字典的时候，结果Series中的索引将是排序后的字典的建。



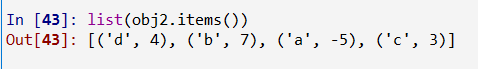
在这种情况下， sdata 中的3个值被放在了合适的位置，但因为没有发现对应于 ‘California’ 的值，就出现了NaN （不是一个数），这在pandas中被用来标记数据缺失或NA 值。我使用“missing”或“NA”来表示数度丢失。

**(2)series对象转换为字典dict**

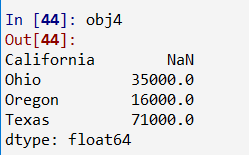
从series的字典构建中可以看出他们互相转换的机制了：将series对象的index作为keys，对应的值作为dict的value。

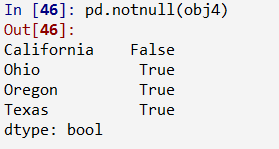
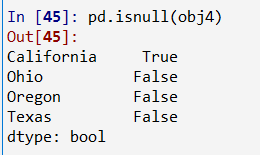
**(3)series对象转换为tuple列表**

list(se.items())

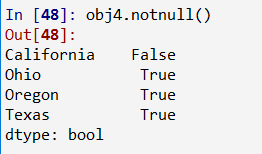
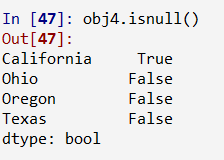


pandas中用**函数isnull 和notnull 来检测数据丢失**：





Series也提供了这些函数的实例方法：



**(4)series对象操作**

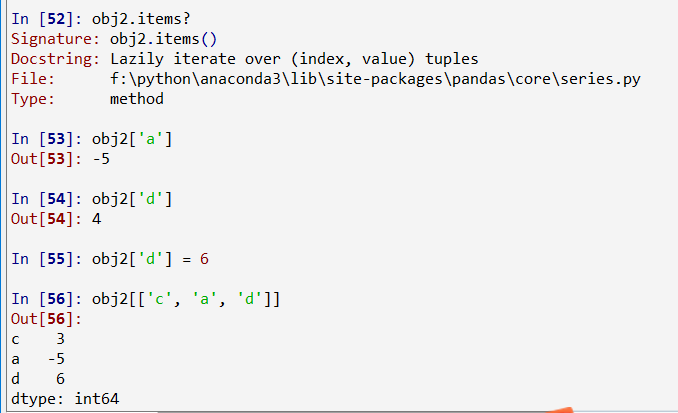
**series对象迭代**

Series.iteritems()

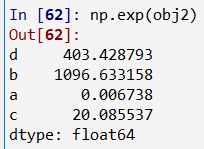
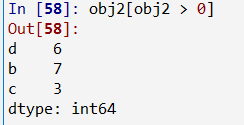
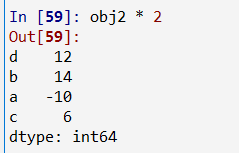
Lazily iterate over (index, value) tuples

[i.split(**','**) *for* \_, i *in* df[**'VenueCategory'**].iteritems()]

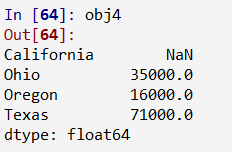
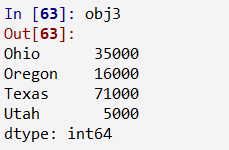
与正规的NumPy数组相比，你可以**使用索引里的值来选择**一个单一值或一个值集：

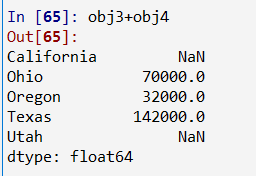


NumPy数组操作，例如通过一个**布尔数组过滤，纯量乘法，使用数学函数**，将会保持索引和值间的关联：

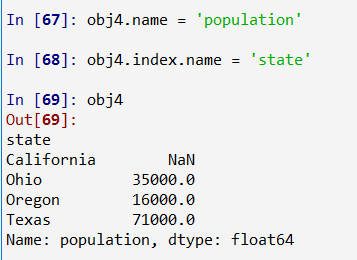
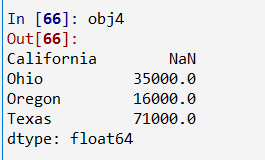


在许多应用中Series的一个重要功能是在**算术运算**中它会自动对齐不同索引的数据：



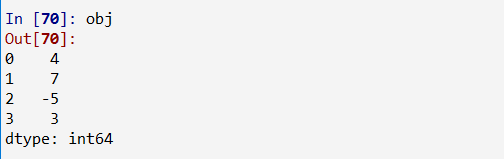


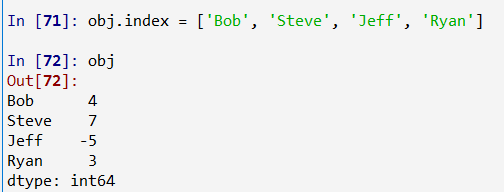
Series对象本身和它的索引都有一个 **name 属性**，它和pandas的其它一些关键功能整合在一起：



**(5)Series索引更改**

可以通过赋值就地更改：



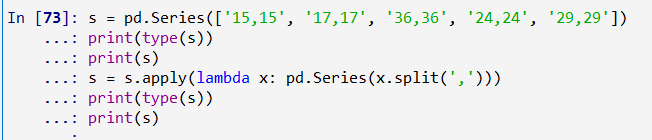


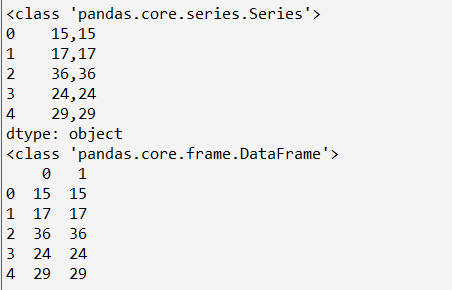
**(6)series值替换**

ser.replace(1, 11)

可以使用字典映射：将1替换为11，将2替换为12  
ser.replace({1:11, 2:12})

**(7)series列分割转换成dataframe**



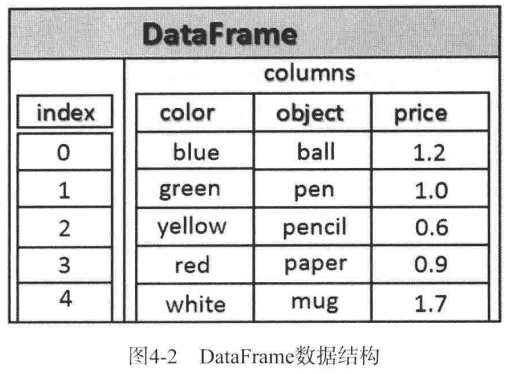


Note: series对象直接应用apply方法是不会改变原series对象的，要赋值修改。

7.1.3 DataFrame

DataFrame是Python中Pandas库中的一种数据结构，它类似excel，是一种**二维表，**DataFrame的单元格可以存放数值、字符串等，这和excel表很像。同时DataFrame可以设置列名**columns**与行名**index**，可以通过像matlab一样通过位置获取数据也可以通过列名和行名定位

DataFrame由按照一定顺序排列的多列数据组成，各列的数据类型可以有所不同。（数值，字符串或布尔值等）



**7.2 pandas读取和保存CSV文件**

**8. json模块**

**9. sys模块**

sys模块主要用于访问与Python解释器联系紧密的变量和函数。

|  |  |
| --- | --- |
| 函数/变量 | 描述 |
| argv | 命令行参数，包括脚本名称 |
| exit([argv]) | 退出当前的程序，可选参数为给定的返回值或者错误信息 |
| modules | 映射模块名字到载入模块的字典 |
| path | 查找模块所在目录的目录名列表 |
| platform | 类似sunos5或者win32的平台标识符 |
| Stdin | 标准输入流 —— 一个类文件（file-like）对象 |
| stdout | 标准输出流 —— 一个类文件对象 |
| Stderr | 标准错误流 —— 一个类文件对象 |

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* **sys.argv**

可以用sys.argv获取当前正在执行的命令行参数的参数列表(list)。

| **变量** | **解释** |  |
| --- | --- | --- |
| sys.argv[0] | 当前程序名 |  |
| sys.argv[0] | 第一个参数 |  |
| sys.argv[1] | 第二个参数 |  |

# encoding: utf-8

# filename: argv\_test.py

import sys

# 获取脚本名字

print('The name of this program is: %s' %(sys.argv[0]))

# 获取参数列表

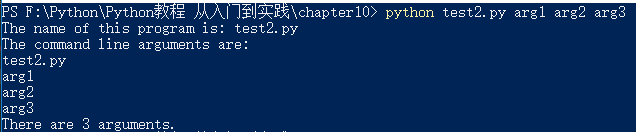
print('The command line arguments are:')

for i in sys.argv:

print i

# 统计参数个数

print 'There are %s arguments.'%(len(sys.argv)-1)



**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

sys.exit(n)

调用sys.exit(n)可以中途退出程序，当参数非0时，会引发一个SystemExit异常，从而可以在主程序中捕获该异常。  
看代码：

import sys

print("running...")

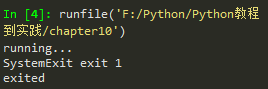
try:

sys.exit(1)

except SystemExit:

print("SystemExit exit 1")

print("exited")



**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* **sys.platform**

获取当前执行环境的平台，如win32表示是Windows 32bit操作系统，linux2表示是linux平台；

# encoding: utf-8

# linux

>>> import sys

>>> sys.platform

'linux2'

# windows

>>> import sys

>>> sys.platform

'win32'

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* **sys.stdout.write**

参考：https://blog.csdn.net/orangleliu/article/details/42915501

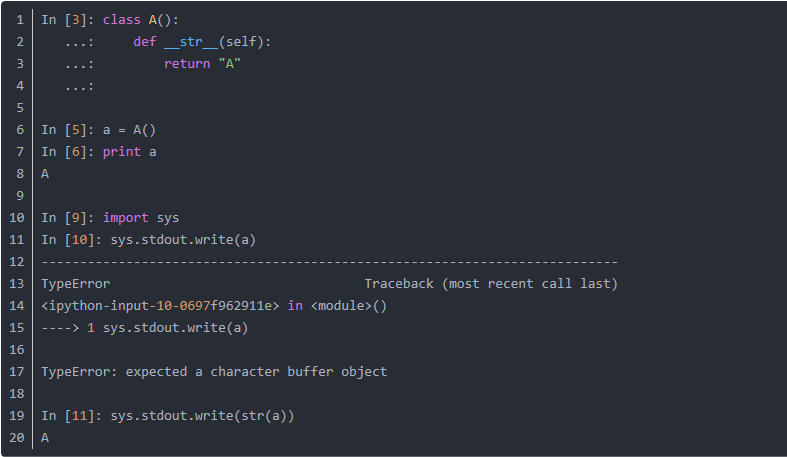
Signature: sys.stdout.write(string)

Docstring:Write string to stream.

Returns the number of characters written (which is always equal to the length of the string).

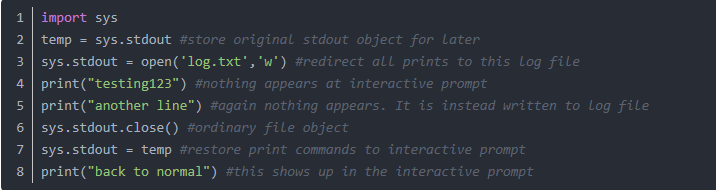
区别：print是对 sys.stdout.write的友好封装

print 可以把一个对象转化成str然后放到标准输出中， sys.stdout.write 需要把对象先转化成对象在输出



所以说不能用 sys.stdout.write来直接代替 print

运用：两者可以结合使用



第2行之所以先把sys.stdout保存为temp，就是因为sys.stdout.write本来的指令就是print, 第三行中将sys.stdout的链接转为打开文件指令，后面写入文件后再关闭后，在使用temp暂存的sys.stdout的原始含义

log.txt文件中保存输出结果为：



我们可以把调试中打印的信息保存的文件中，这样也是追中和查找错误的一个方式

还有就是多线程日志中可能会用到 sys.stdout.write 来封装日志写入

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* **sys.stdout.flush()**

这里说的区别是针对python2的，在python中没有区别：



这个程序本意是每隔一秒输出一个数字，但是如果把这句话sys.stdout.flush()注释的话，你就只能等到程序执行完毕，屏幕上会一次性输出0，1，2，3，4。

如果你加上sys.stdout.flush()，刷新stdout，这样就能每隔一秒输出一个数字了。

可以用在网络程序中多线程程序，多个线程后台运行，同时要能在屏幕上实时看到输出信息。

# [10. argparse模块](https://docs.python.org/3/library/argparse.html#module-argparse)

python中的命令行解析最简单最原始的方法是使用sys.argv来实现，但是大多数情况下，脚本很可能需要多个参数，而且每次参数的类型用处各不相同，那么这个时候在参数前添加标签表明参数的类型和用途便十分有用，而利用argparse模块可以很方便得实现这一目的。

基本使用

import argparse #首先导入模块

parser=argparse.ArgumentParser() #创建一个解析对象

parser.add\_argument("echo",help="echo the string") #向该对象中添加你要关注的命令行参数和选项

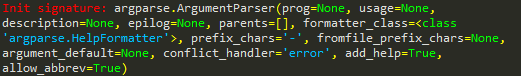
args=parser.parse\_args() #进行解析

print(args.echo)

现在对argparse的每一个模块进行介绍：

**10.1 argparse.ArgumentParser**

创建解析器对象ArgumentParser**，**用于将命令行字符串解析为Python对象。



参数解释：

Keyword Arguments:

- prog – 程序名 (default: sys.argv[0])

- usage – 用户信息 (default: auto-generated from arguments)

- description -- - help时显示的开始文字

- epilog --  - help时显示的结尾文字

- parents -- Parsers whose arguments should be copied into this one

- formatter\_class -- HelpFormatter class for printing help messages

- prefix\_chars --  - 命令的前缀，默认是‘-’

- fromfile\_prefix\_chars -- Characters that prefix files containing additional arguments 命令行参数从文件中读取

- argument\_default – 参数默认值 The default value for all arguments

- conflict\_handler – 处理冲突String indicating how to handle conflicts

- add\_help – 增加帮助选项 Add a -h/-help option

- allow\_abbrev – 缩写Allow long options to be abbreviated unambiguously

[**10.2 parser.add\_argument**](https://www.cnblogs.com/dengtou/p/8413609.html)

向该对象中添加你要关注的命令行参数和选项



定位参数：

parser.add\_argument(dest=’echo’ ,help="echo the string")

也可以认为是name or flags,  - 必选，指定参数的形式，一般写两个，一个短参数，一个长参数

可选参数：

parser.add\_argument(option\_string="—verbosity", help="increase output verbosity")

在执行程序的时候，定位参数必选，可选参数可选。

[add\_argument()常用的参数：](https://www.cnblogs.com/zknublx/p/6106343.html)

dest：如果提供dest，例如dest="a"，那么可以通过args.a访问该参数（见案例6）

default：设置参数的默认值

action：参数出发的动作 #表示值赋予键的方式，action意思是当读取的参数中出现指定参数的时候的行为

choices    - 设置参数的范围，如果choice中的类型不是字符串，要指定type. 表示该参数能接受的值只能来自某几个值候选值中，除此之外会报错，用choice参数即可

nargs    - 指定这个参数后面的value有多少个，默认为1

dest   - 设置这个选项的value解析出来后放到哪个属性中

store：保存参数，默认

store\_const：保存一个被定义为参数规格一部分的值（常量），而不是一个来自参数解析而来的值。

store\_ture/store\_false：保存相应的布尔值

append：将值保存在一个列表中。

append\_const：将一个定义在参数规格中的值（常量）保存在一个列表中。

count：参数出现的次数

parser.add\_argument("-v", "--verbosity", action="count", default=0, help="increase output verbosity")

version：打印程序版本信息

metavar - A name for the argument in usage messages.

required: 决定这个参数是不是必须参数的选项，通常设置的参数都是可选的，就是你可以定以后但不赋予程序这个值，但设置成required=True时，这个参数就是必须要赋值的了。

type：把从命令行输入的结果转成设置的类型

choice：允许的参数值。设置参数的范围，如果choice中的类型不是字符串，要指定type表示该参数能接受的值只能来自某几个值候选值中，除此之外会报错，用choice参数即可

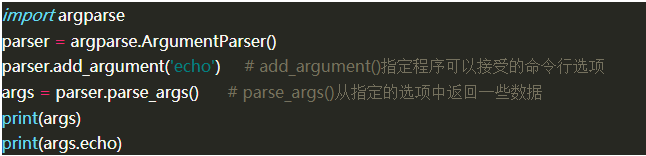
nargs: 指定这个参数后面的value有多少个，默认为1. 当选项后接受多个或者0个参数时需要这个来指定 (见实验5)

#\*\* parser.add\_argument("-v", "--verbosity", type=int, choices=[0, 1, 2], help="increase output verbosity")

help：参数命令的介绍

name or flags...    - 必选，指定参数的形式，一般写两个，一个短参数，一个长参数

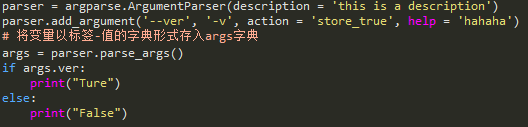
案例1：



输出结果：

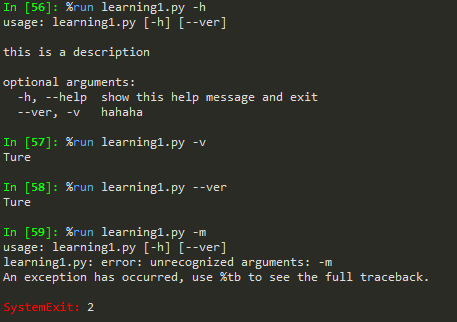


案例2：对action和name/flags参数的解释。当使用action时，可以%run learning1.py –v 而不用往里面输入参数，但是不使用action时，要么是不写-v,要是写-v就必须输入参数



从这个例子可以看出，name/flags设置成—ver时，获取参数值可以args.ver

结果：

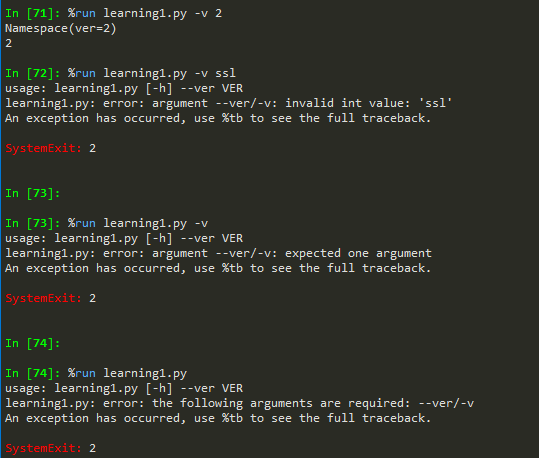


对required，type，choices三个参数的详解：

案例3：



结果：

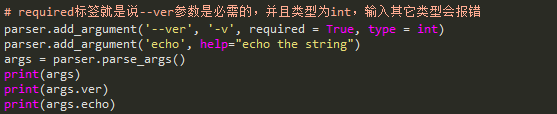


分析：第一和第四个结果说明了这个参数v是必须的，这也是required的缘故。

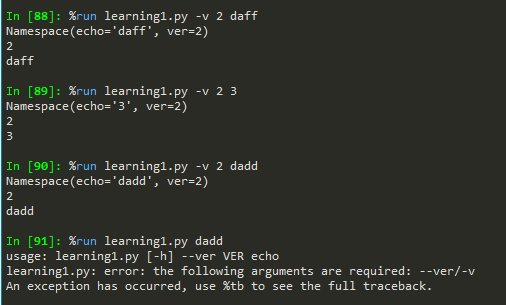
第一和第二个结果说明了type的用处

第一和第三个结果说明了出现-v不赋值也不出现bug是action的结构

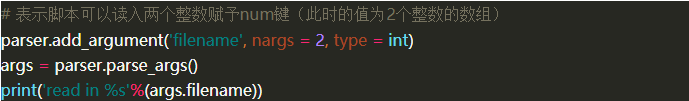
案例4：echo是必选参数，可以不写，但是只要写上，就一定要输入参数，而—ver可以说是name/flags,不一定需要赋值，加上required后就必须赋值



结果：



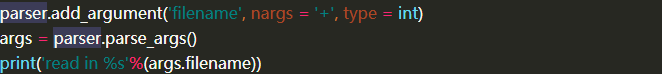
案例5： nargs - 指定这个参数后面的value有多少个，默认为1



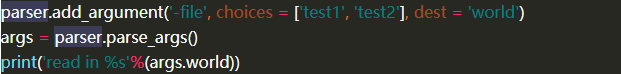
结果



分析：nargs还可以'\*'用来表示如果有该位置参数输入的话，之后所有的输入都将作为该位置参数的值；‘+’表示读取至少1个该位置参数。'?'表示该位置参数要么没有，要么就只要一个。（PS：跟正则表达式的符号用途一致。）



案例6： dest   - 设置这个选项的value解析出来后放到哪个属性中



结果：

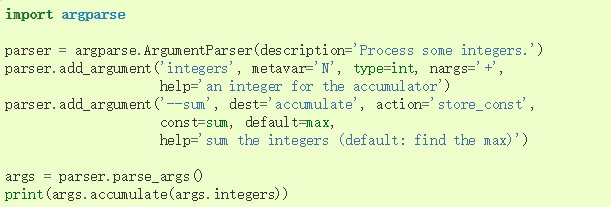


**10.3 parser.parse\_args**

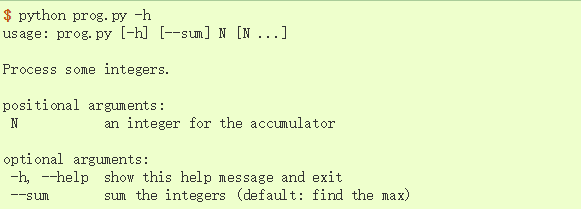
对解析对象进行Signature: parser.parse\_args(args=None, namespace=None)

**总体案例：**

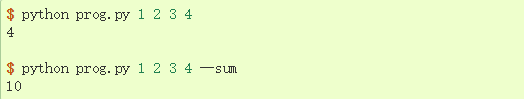
以下代码是一个Python程序，它获取整数列表并生成sum或max：



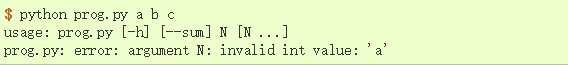
假设上面的Python代码保存到一个名为的文件中prog.py，它可以在命令行运行并提供有用的帮助消息：



使用适当的参数运行时，它会输出命令行整数的总和或最大值：



如果传入无效参数，则会发出错误：



来源：https://docs.python.org/3/library/argparse.html#module-argparse

**11. inspect模块**

* [1. 访问对象的属性](https://www.cnblogs.com/mosson/p/7244480.html#_label0)
* [2. 访问对象的元数据](https://www.cnblogs.com/mosson/p/7244480.html#_label1)
* [3. 使用inspect模块](https://www.cnblogs.com/mosson/p/7244480.html#_label2)

inspect模块主要提供了四种用处：

(1).对是否是模块，框架，函数等进行类型检查。

(2).获取源码

(3).获取类或函数的参数的信息

(4).解析堆栈

使用inspect模块可以提供自省功能，下面是关于自省的一些介绍:

# encoding: utf-8

import sys

import inspect

# 定义备用函数

def foo(): pass

class Cat():

def \_\_init\_\_(self,name='kitty'):

self.name = name

def sayHi(self):

print(self.name + "say Hi!")

cat = Cat('kucy')

try:

print(Cat.sayHi())

except:

print('The argument is not useful')

cat.sayHi()



**11.1 访问对象的属性**

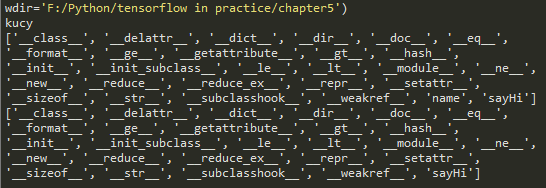
以下列出了几个内建方法，可以用来检查或是访问对象的属性。这些方法可以用于任意对象而不仅仅是例子中的Cat实例对象；Python中一切都是对象。

cat = Cat('kucy')

print(cat.name)

print(dir(cat))

print(dir(Cat))



由此可以看出，self.name将name定义为了Cat中的属性，可以直接通过Class.name获得其值。

再看内置函数dir来获取访问对象的属性，对cat，因为给实例赋予了实值，所以就比Cat类多了个name属性。

cat = Cat('kucy')

if hasattr(cat, 'name'): # 检查obj是否有一个名为attr的值的属性，返回一个布尔值

setattr(cat, 'name','tiger') # 给obj的名为attr的值的属性赋值为val。

print(getattr(cat,'name')) # 返回obj中名为attr值的属性的值

getattr(cat,'sayHi')() # 返回obj中名为attr值的属性的值，若attr为函数，要在后面再加个()



* **dir([obj]):**   
  调用这个方法将返回包含obj大多数属性名的列表（会有一些特殊的属性不包含在内）。obj的默认值是当前的模块对象。
* **hasattr(obj, attr):**这个方法用于检查obj是否有一个名为attr的值的属性，返回一个布尔值。
* **getattr(obj, attr):**调用这个方法将返回obj中名为attr值的属性的值，例如如果attr为'bar'，则返回obj.bar。
* **setattr(obj, attr, val):**   
  调用这个方法将给obj的名为attr的值的属性赋值为val。例如如果attr为'bar'，则相当于obj.bar = val。

**11.2 访问对象的元数据**

1x. Python内置函数和语句

**1x.1 assert断言**

["Assert statements are a convenient way to insert debugging assertions into a program".](https://www.cnblogs.com/hezhiyao/p/7805278.html)

assert condition

用来让程序测试这个condition，如果condition为false，那么raise一个AssertionError出来。逻辑上等同于：

if not condition:

raise AssertionError()

例子：

>>> assert 1==0

Traceback (most recent call last):

File "<pyshell#1>", line 1, in <module>

assert 1==0

AssertionError

>>> assert True

>>> assert False

Traceback (most recent call last):

File "<pyshell#3>", line 1, in <module>

assert False

AssertionError

>>> assert 3<2

Traceback (most recent call last):

File "<pyshell#4>", line 1, in <module>

assert 3<2

AssertionError

如何为assert断言语句添加异常参数

assert的异常参数，其实就是在断言表达式后添加字符串信息，用来解释断言并更好的知道是哪里出了问题。格式如下：

assert expression [, arguments]

assert 表达式 [, 参数]

比如如下的例子：

>>> assert len(lists) >=5,'列表元素个数小于5'

Traceback (most recent call last):

File "D:/Data/Python/helloworld/helloworld.py", line 1, in <module>

assert 2>=5,'列表元素个数小于5'

AssertionError: 列表元素个数小于5

>>> assert 2==1,'2不等于1'

Traceback (most recent call last):

File "D:/Data/Python/helloworld/helloworld.py", line 1, in <module>

assert 2==1,'2不等于1'

AssertionError: 2不等于1

**1x.2 dir()函数**

dir() 函数不带参数时，返回当前范围内的**变量、方法和定义**的**类型列表**；带参数时，返回参数的属性、方法列表。如果参数包含方法\_\_dir\_\_()，该方法将被调用。如果参数不包\_\_dir\_\_()，该方法将最大限度地收集参数信息。

**dir 语法：**

dir([object])

* 参数说明：

object -- 对象、变量、类型。

* 返回值：

返回模块的属性列表。

>>> assert 2==1,'2不等于1'

>>>dir() # 获得当前模块的属性列表

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'arr', 'myslice']

>>>dir([])或者是dir(list) # 查看list包含的方法

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

**1x.3 \_\_all\_\_属性**

* 在模块(\*.py)中使用意为导出\_\_all\_\_列表里的类、函数、变量等成员，

否则将导 modualA中所有不以下划线开头（带下划线的是只能供这个类使用的）的成员，在模块中使用\_\_all\_\_属性可避免在相互引用时的命名冲突

modualA.py

# \_\_all\_\_ 限定了模块A中只能导出函数"fun1",和类"class1”  
\_\_all\_\_=["fun1","class1"]

...

#end file modualA.py

使用：  
from modualA import \*  
导入模块modualA里的所有成员（如果定义了\_\_all\_\_那么就导出列表中的所有（只能导出函数"fun1",和类"class1”），否则默认导出不以下划线开头的所有成员）

**测试实验：**

foo.py文件

\_\_all\_\_ = ['bar', 'baz']

waz = 5

bar = 10

def baz():

return 'baz'

创建run-foo.py引用上述文件

# -\*- coding: utf-8 -\*-

from foo import \*

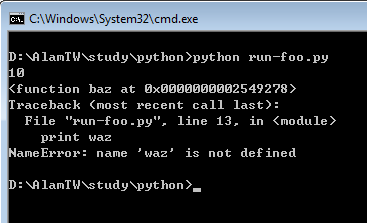
print bar

print baz

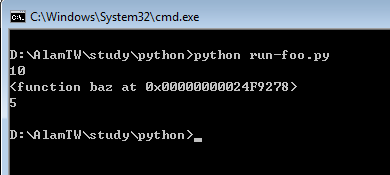
# The following will trigger an exception, as "waz" is not exported by the module

# 下面的代码就会抛出异常，因为 "waz"并没有从模块中导出，因为 \_\_all\_\_ 没有定义

print waz



把foo.py的“\_\_all\_\_ = ['bar', 'baz']” 注释，运行正常



它不仅在第一时间展现了模块的内容大纲，而且也更清晰的提供了外部访问接口。

* 包含所有模块文件的文件夹pkgA的\_\_init\_\_.py中意为导出包里的模块

pkgA/\_\_init\_\_.py

# 只是用pkgA文件夹中的"modualA","modualB"两个模块（文件）

\_\_all\_\_=["modualA","modualB"]

from modualA import class1,class2

from modualB import fun1,class3

....

#end file pkgA/\_\_init\_\_.py

使用：

from pkgA import \*

以上语句即执行了pkgA下的\_\_init\_\_.py，导入两个模块，和这两模块下的函数和类

**1x.4 下划线的属性**

\_\_foo\_\_:一种约定,Python内部的名字,用来区别其他用户自定义的命名,以防冲突.

\_foo:一种约定,用来指定变量私有.程序员用来指定内部变量的一种方式.(你可以访问，但是不要轻易访问。)

\_\_foo:这个有真正的意义:解析器用\_classname\_\_foo来代替这个名字,以区别和其他类相同的命名. 也是可以访问，不过需要.class.\_\_foo。（python对类定义里的两个以下划线开头《但不以两个下划线结尾》的名字做了特殊处理，使得在类定义之外不能直接使用这个名字访问，起到保护的作用）

Python docs的说明，\_object和\_\_object的作用域限制在本模块内。

1、\_xxx 不能用于’from module import \*’ 以单下划线开头的表示的是protected类型的变量。即保护类型只能允许其本身与子类进行访问。

2、\_\_xxx 双下划线的表示的是私有类型的变量。只能是允许这个类本身进行访问了。连子类也不可以

3、\_\_xxx\_\_\_ 定义的是特列方法。像\_\_init\_\_之类的

**1x.5 @修饰符**

**简单的说，@装饰器就是用来提供调用的，**

def funA(arg):

print 'A'

a=arg()

@funA

def funB():

print 'B'

输出结果为：



此处的@相当于funA(funB())

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

装饰器的语法以@开头，接着是装饰器函数的名字和可选的参数。紧跟着装饰器声明的是被修饰的函数，和修饰函数的可选参数。

class MyClass(obj):

@staticmethod

def staticFoo():

...

**利用staticmethod内建函数来将这个函数“转化”为静态方法，利用装饰器会很简洁。**

（可以参见《数据结构与算法 python语言描述》P41）

@g

@f

def foo():

...

相当于foo = g(f(foo()))

**有参数的装饰器：**

@decomaker(deco\_args)

deffoo():

pass

需要自己返回以函数作为参数的装饰器，换句话说，decomaker()用deco\_args做了些事情并返回函数对象，而该函数正是以foo作为其参数的装饰器，相当于：   
foo=decomaker(deco\_args)(foo)

多个装饰器的例子，其中一个装饰器带有一个参数：

@deco1(deco\_args)

@deco2

def func():pass

等价于：   
func=deco1(deco\_args) (deco2(func))

**1x.6 \*的用法**

这里参考的是：https://www.cnblogs.com/jony7/p/8035376.html

1.表示乘号



2.“\*\*”表示指数相乘

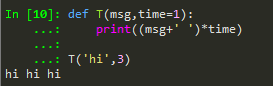


3.表示倍数

def T(msg,time=1):

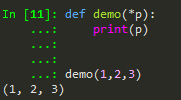
print((msg+' ')\*time)

T('hi',3)



4. 单个的\*的两种用法：

（1） \*parameter是用来接受任意多个参数并将其放在一个元组中



def multiple(arg, \*args):

print("arg: ", arg)

#打印不定长参数

for value in args: #至于为什么能用for遍历，是因为args是一个元组！

print("other args:", value)

if \_\_name\_\_ == '\_\_main\_\_':

multiple(1,'a',True)



（2）函数在调用多个参数时，在列表、元组、集合、字典及其他可迭代对象作为实参，并在前面加 \*

如 \*（1,2,3）解释器将自动进行解包然后传递给多个单变量参数（参数个数要对应相等）。

>>> def d(a,b,c):  
    print(a,b,c)

>>> d(1,2,3)  
1 2 3  
>>> a=[1,2,3]  
>>> b=[1,2,3]  
>>> c=[1,2,3]  
>>> d(a,b,c)  
[1, 2, 3] [1, 2, 3] [1, 2, 3]

>>> d(\*a)  
1 2 3

5. 加了两个\*号（\*\*）的变量名会存放所有未命名的变量参数（以dict字典的形式存储）

def multiple2(\*\*args):

#打印不定长参数

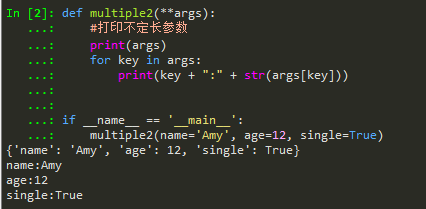
print(args)

for key in args:

print(key + ":" + str(args[key]))

if \_\_name\_\_ == '\_\_main\_\_':

multiple2(name='Amy', age=12, single=True)



def demo(\*\*p):

for i in p.items():

print(i)

demo(x=1,y=2)



**一个综合的例子：**

def multiple(arg, \*args, \*\*dictargs):

print "arg: ", arg

#打印args

for value in args:

print "other args:", value

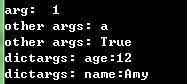
#打印dict类型的不定长参数 args

for key in dictargs:

print "dictargs:" + key + ":" + bytes(dictargs[key])

if \_\_name\_\_ == '\_\_main\_\_':

multiple(1,'a',True, name='Amy',age=12, )



**1x.7 yield的用法**

您可能听说过，带有 yield 的函数在 Python 中被称之为 generator（生成器），何谓 generator ？

我们先抛开 generator，以一个常见的编程题目来展示 yield 的概念。

**如何生成斐波那契数列**

斐波那契（Fibonacci）数列是一个非常简单的递归数列，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。用计算机程序输出斐波那契數列的前 N 个数是一个非常简单的问题，许多初学者都可以轻易写出如下函数：

**清单 1. 简单输出斐波那契數列前 N 个数**

def fab(max):

n, a, b = 0, 0, 1

while n < max:

print b

a, b = b, a + b

n = n + 1

fab(5)

执行以上代码，我们可以得到如下输出：

1

1

2

3

5

结果没有问题，但有经验的开发者会指出，直接在 fab 函数中用 print 打印数字会导致该函数可复用性较差，因为 fab 函数返回 None，其他函数无法获得该函数生成的数列。

要提高 fab 函数的可复用性，最好不要直接打印出数列，而是返回一个 List。以下是 fab 函数改写后的第二个版本：

**清单 2. 输出斐波那契數列前 N 个数第二版**

def fab(max):

n, a, b = 0, 0, 1

L = []

while n < max:

L.append(b)

a, b = b, a + b

n = n + 1

return L

for n in fab(5): print n

可以使用如下方式打印出 fab 函数返回的 List：

1

1

2

3

5

改写后的 fab 函数通过返回 List 能满足复用性的要求，但是更有经验的开发者会指出，该函数在运行中占用的内存会随着参数 max 的增大而增大，如果要控制内存占用，最好不要用 List来保存中间结果，而是通过 iterable 对象来迭代。例如，在 Python2.x 中，代码：

**清单 3. 通过 iterable 对象来迭代**

for i in range(1000): pass

会导致生成一个 1000 个元素的 List，而代码：

for i in xrange(1000): pass

则不会生成一个 1000 个元素的 List，而是在每次迭代中返回下一个数值，内存空间占用很小。因为 xrange 不返回 List，而是返回一个 iterable 对象。

利用 iterable 我们可以把 fab 函数改写为一个支持 iterable 的 class，以下是第三个版本的 Fab：

**清单 4. 第三个版本**

class Fab(object):

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

self.max = max

self.n, self.a, self.b = 0, 0, 1

def \_\_iter\_\_(self):

return self

def next\_\_(self):

if self.n < self.max:

r = self.b

self.a, self.b = self.b, self.a + self.b

self.n = self.n + 1

return r

raise StopIteration()

for n in Fab(5): print n

Fab 类通过 next() 不断返回数列的下一个数，内存占用始终为常数：

1

1

2

3

5

然而，使用 class 改写的这个版本，代码远远没有第一版的 fab 函数来得简洁。如果我们想要保持第一版 fab 函数的简洁性，同时又要获得 iterable 的效果，yield 就派上用场了：

**清单 5. 使用 yield 的第四版**

**实例**

def fab(max):

n, a, b = 0, 0, 1

while n < max:

yield b # 使用 yield

# print b

a, b = b, a + b

n = n + 1

for n in fab(5): print n

第四个版本的 fab 和第一版相比，仅仅把 print b 改为了 yield b，就在保持简洁性的同时获得了 iterable 的效果。

调用第四版的 fab 和第二版的 fab 完全一致：

1

1

2

3

5

简单地讲，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，调用 fab(5) 不会执行 fab 函数，而是返回一个 iterable 对象！在 for 循环执行时，每次循环都会执行 fab 函数内部的代码，执行到 yield b 时，fab 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。

也可以手动调用 fab(5) 的 next() 方法（因为 fab(5) 是一个 generator 对象，该对象具有 next() 方法），这样我们就可以更清楚地看到 fab 的执行流程：

**清单 6. 执行流程**

>>> f = fab(5)

>>> f.next() 1

>>> f.next() 1

>>> f.next() 2

>>> f.next() 3

>>> f.next() 5

>>> f.next()

Traceback (most recent call last): File "<stdin>", line 1, in <module> StopIteration

当函数执行结束时，generator 自动抛出 StopIteration 异常，表示迭代完成。在 for 循环里，无需处理 StopIteration 异常，循环会正常结束。

**我们可以得出以下结论：**

一个带有 yield 的函数就是一个 generator，它和普通函数不同，生成一个 generator 看起来像函数调用，但不会执行任何函数代码，直到对其调用 next()（在 for 循环中会自动调用 next()）才开始执行。虽然执行流程仍按函数的流程执行，但每执行到一个 yield 语句就会中断，并返回一个迭代值，下次执行时从 yield 的下一个语句继续执行。看起来就好像一个函数在正常执行的过程中被 yield 中断了数次，每次中断都会通过 yield 返回当前的迭代值。

yield 的好处是显而易见的，把一个函数改写为一个 generator 就获得了迭代能力，比起用类的实例保存状态来计算下一个 next() 的值，不仅代码简洁，而且执行流程异常清晰。

如何判断一个函数是否是一个特殊的 generator 函数？可以利用 isgeneratorfunction 判断：

**清单 7. 使用 isgeneratorfunction 判断**

>>> from inspect import isgeneratorfunction

>>> isgeneratorfunction(fab) True

要注意区分 fab 和 fab(5)，fab 是一个 generator function，而 fab(5) 是调用 fab 返回的一个 generator，好比类的定义和类的实例的区别：

**清单 8. 类的定义和类的实例**

>>> import types

>>> isinstance(fab, types.GeneratorType) False

>>> isinstance(fab(5), types.GeneratorType) True

fab 是无法迭代的，而 fab(5) 是可迭代的：

>>>from collections import Iterable

>>> isinstance(fab, Iterable) False

>>> isinstance(fab(5), Iterable) True

每次调用 fab 函数都会生成一个新的 generator 实例，各实例互不影响：

>>>f1 = fab(3)

>>> f2 = fab(5)

>>> print 'f1:', f1.next() f1: 1

>>> print 'f2:', f2.next() f2: 1

>>> print 'f1:', f1.next() f1: 1

>>> print 'f2:', f2.next() f2: 1

>>> print 'f1:', f1.next() f1: 2

>>> print 'f2:', f2.next() f2: 2

>>> print 'f2:', f2.next() f2: 3

>>> print 'f2:', f2.next() f2: 5

**return 的作用**

在一个 generator function 中，如果没有 return，则默认执行至函数完毕，如果在执行过程中 return，则直接抛出 **StopIteration** 终止迭代。

**另一个例子**

另一个 yield 的例子来源于文件读取。如果直接对文件对象调用 read() 方法，会导致不可预测的内存占用。好的方法是利用固定长度的缓冲区来不断读取文件内容。通过 yield，我们不再需要编写读文件的迭代类，就可以轻松实现文件读取：

**清单 9. 另一个 yield 的例子**

**实例**

def read\_file(fpath):

BLOCK\_SIZE = 1024

with open(fpath, 'rb') as f:

while True:

block = f.read(BLOCK\_SIZE)

if block:

yield block

else: return

以上仅仅简单介绍了 yield 的基本概念和用法，yield 在 Python 3 中还有更强大的用法，我们会在后续文章中讨论。

**1x.8 lambda的用法**

lambda表达式，通常是在**需要一个函数，但是又不想费神去命名一个函数**的场合下使用，也就是指**匿名函数**。

lambda所表示的匿名函数的内容应该是很简单的，如果复杂的话，干脆就重新定义一个函数了，使用lambda就有点过于执拗了。

lambda就是用来定义一个匿名函数的，如果还要给他绑定一个名字的话，就会显得有点画蛇添足，通常是直接使用lambda函数。如下所示：

add = lambda x, y : x+y

add(1,2) # 结果为3

**1x.9** [**enumerate详解**](https://www.cnblogs.com/huangbiquan/p/7854246.html)

enumerate()是python的内置函数，在字典上是枚举、列举的意思  
enumerate参数为可遍历/可迭代的对象(如列表、字符串)  
enumerate多用于在for循环中得到计数，利用它可以同时获得索引和值，即需要index和value值的时候可以使用enumerate  
enumerate()返回的是一个enumerate对象

>>> lst = [1, 2, 3, 4, 10, 5]

>>> enumerate(lst)

<enumerate object at 0x00000000032A3990>

>>> list(enumerate(lst))

[(0, 1), (1, 2), (2, 3), (3, 4), (4, 10), (5, 5)]

enumerate的使用：  
例如：已知lst = [1,2,3,4,5,6]，要求输出:  
0,1  
1,2  
2,3  
3,4  
4,5  
5,6

在for循环中使用enumerate，同时得到索引和值是很常见的用法

>>> lst = [1,2,3,4,5,6]

>>> for index,value in enumerate(lst):

print ('%s,%s' % (index,value))

0,1

1,2

2,3

3,4

4,5

5,6

#指定索引从1开始

>>> lst = [1,2,3,4,5,6]

>>> for index,value in enumerate(lst,1):

print ('%s,%s' % (index,value))

1,1

2,2

3,3

4,4

5,5

6,6

#指定索引从3开始

>>> for index,value in enumerate(lst,3):

print ('%s,%s' % (index,value))

3,1

4,2

5,3

6,4

7,5

8,6

enumerate实战

line 是个 string 包含 0 和 1，要把1都找出来：

*#方法一*

def read\_line(line):

sample = {}

n = **len**(line)

for i in **range**(n):

if line[i]!='0':

sample[i] = **int**(line[i])

return sample

*#方法二*

def xread\_line(line):

return((idx,**int**(val)) for idx, val in **enumerate**(line) if val != '0')

print read\_line('0001110101')

print **list**(xread\_line('0001110101'))

补充：  
如果要统计文件的行数，可以这样写：  
count = len(open(filepath, 'r').readlines())  
这种方法简单，但是可能比较慢，当文件比较大时甚至不能工作。

可以利用enumerate()：  
count = 0  
for index, line in enumerate(open(filepath,'r'))：   
   count += 1