**import的用法：**

如果你已经把my\_abs()的函数定义保存为abstest.py文件了，那么，可以在该文件的当前目录下启动Python解释器，用from abstest import my\_abs来导入my\_abs()函数，注意abstest是文件名（不含.py扩展名）

main.py与showinfo.py同级

showinfo.py定义了两个函数，showself(),showother()

main.py想使用showinfo.py中的函数，则

(1) from showinfo import showself,showother ，在main.py中直接showself()

(2) 或者from showinfo import\* ，在main.py中直接showself()

(3)或者import showinfo，在main.py中showinfo .showself()

main.py与文件夹next同级，next文件夹下有praiseinfo.py文件，定义了praiseother()和praiseself()：

或者

from next.praiseinfo import praiseself,praiseother，在main.py中

praiseother()

praiseself()

或者：

from next import praiseinfo，在main.py中

praiseinfo.praiseother()

praiseinfo.praiseself()

**python作用域：**

python的作用域由def、class、lambda等语句产生，if、try、for等语句并不会产生新的作用域，换句话说就是if、try、for等语句里面定义的变量并不会随着该语句执行结束而回收，而是可以到处引用的，if、try、for等语句内变量的作用域实际范围受其所在的def、class、lambda范围约束。

*def* main():

    if False:

        desc='bad'

    else:

        desc='ok'

    print(desc)  #ok

在表达式中引用变量时，Python解释器将按如下顺序遍历各作用域，以解析该引用：

1)当前函数作用域。

2)任何外围作用域（例如，包含当前函数的上层函数)

3)包含当前代码的那个模块的作用域（也叫全局作用域，global scope)

4)内置作用域（即包含len()及str()等Python内置函数的那个作用域)

5)如果上面这些地方都没有定义过名称相符的变量，那就抛出NameError异常。

给变量赋值时，规则有所不同：

如果当前作用域内已经定义了这个变量，那么该变量就会具备新值。

若是当前作用域内没有这个变量，Python则会把这次赋值视为对该变量的定义。而新定义的这个变量，其作用域就是包含赋值操作的这个函数。

（1）先赋值再访问

success=False

*def* cal():

    if True:

        success=True

    else:

        success=False

    print(success)

cal() #True

print(success)  #false

（2）先访问再赋值

varname = "ok"

*def* func():

    print(varname) #报错： local variable 'varname' referenced before assignment

    varname = "wrong"

func()

print(varname)

重名时，如果要指定使用上层作用域的变量，则使用nolocal或global声明

a=60

*def* func():

    a=60

*def* cb1():

        nonlocal a

        a=100

*def* cb2():

        global a

        a=0

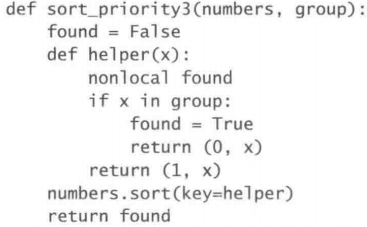
    cb1()

    cb2()

    print(a) #100

func()

print(a) #0

示例：对number排序，而且在group列表中的值要优先于不在group中的

注：Python使用特殊的规则来比较两个元组。它首先比较各元组中下标为0的对应元素，如果相等，再比较下标为1的对应元素，如果还是相等，那就继续比较下标为2的对应元素，依次类推。

三目条件判断：

# c++中bool ret=3>1:true?false

ret=True if 3>1 else False

print(ret)

给python程序传入参数：

import sys

if(len(sys.argv)!=3):

print("input error\n")

exit()

one=sys.argv[1]

another=sys.argv[2]

res=float(one)+float(another)

print("added, result is: %f"%res)

显式传入的参数依次对应argv[1]、argv[2]……，为string

python add.py 3.5 6.7

added, result is: 10.200000

None、False、空字符串、空列表：

在Python中，None、空列表[]、空字典{}、空元组()、0等一系列代表空和无的对象会被转换成False。除此之外的其它对象都会被转化成True

>>> type(None)

<class 'NoneType'>

空值是Python里一个特殊的值，用None表示。None不能理解为0，因为0是有意义的，而None是一个特殊的空值。可以将None赋值给任何变量。

要判断一个变量是否为None，直接使用if a==None:

is与==用于比较：

is比较的是id是不是一样，==比较的是值是不是一样。

Python中，万物皆对象

每个对象包含3个属性，id，type，value。id就是对象地址，可以通过内置函数id()查看对象引用的地址。type就是对象类型，可以通过内置函数type()查看对象的类型。value就是对象的值。

a = 1

b = a

c = 1

d = 1.0

这里有3个对象a、b、c，类型都是int。值都是1。

id(a) # 35556792L

id(b) # 35556792L

id(c) # 35556792L

id(d) # 21253459L

本例中a、b、c都是引用的同一个地址35556792L下的内容。判断a is b的时候，实际上比较的是id(a)==id(b)，结果为True。当你写成a==b时候，实际上比较的是id(a)这个地址指向的值是不是和id(b)这个地址指向值一样，结果为True。同理，a is d的结果就是False。而此时，a==d，结果却是True，因为值是一样的

isinstance(object, type)

其中，object是变量，type是类型(如tuple,dict,int,float,str,list,bool,set等)和class类

若参数object是classinfo类的实例，或者object是classinfo类的子类的一个实例，返回True。若object不是一个给定类型的的对象，则返回False

print(isinstance([1,2,3],*list*)) #true

*class* A:

    pass

*class* B(*A*):

    pass

print(isinstance(B(),A)) #true

a=*bytearray*(2)

a[0]=97

a[1]=*0x*62

print(isinstance(a,*bytes*))  #False

print(isinstance(a,*bytearray*)) #True

python的内置数据类型

python的内置数据类型有：bool int float str list tuple dict set

a=[]

b=()

c={} # or {'name':'liming','age':21}

d={1,3}

e=0

f=0.0

g='hello'

h=False

print(*type*(a))  #<class 'list'>

print(*type*(b))  #<class 'tuple'>

print(*type*(c))  #<class 'dict'>

print(*type*(d))  #<class 'set'>

print(*type*(e))  #<class 'int'>

print(*type*(f))  #<class 'float'>

print(*type*(g))  #<class 'str'>

print(*type*(h))  #<class 'bool'>

set和dict：

创建set，提供一个list或tuple作为set()参数

mySet1=set((1,2,3))

mySet2=set([1,2,3])

或者

s={1,3,5,7}

print(s) #{1, 3, 5, 7}

print(*type*(s)) #<class 'set'>

set中key如果有重复，则重复元素自动被过滤

两个set可以求&、求|

创建dict，提供一个元素为tuple(或list，len必须为2)的tuple(或list)作为dict()参数：

a=[('zcj',21),('lxy',20)]

d=*dict*(a)

a=[['zcj',21],['lxy',20]]

d=*dict*(a)

print(d) #{'zcj': 21, 'lxy': 20}

a=[['zcj',[60,61]],['lxy',[70,71]]]

d=*dict*(a)

print(d) #{'zcj': [60, 61], 'lxy': [70, 71]}

a=[('zcj',21),('lxy',20)]

for k,v in a:

    print(k,':',v,*end*=" ") #zcj : 21 lxy : 20

或者

myDict4={"name":"zhang",'score':100,'weight':75,'name':'wang'};  
print(myDict4); #{'name': 'wang', 'score': 100, 'weight': 75}

dict中key如果有重复，但key对应的value不同，则只保留后边的key-value对

set和dict的key只能是不可变参数，如int、str

因为无法判断两个可变对象是否相等，也就无法保证set\dict内部“不会有重复元素”。

在Python中，字符串、整数等都是不可变的，因此，可以放心地作为key。而list是可变的，就不能作为key

C++的map、set、multimap、multiset的比较函数以及sort算法的比较函数要求严格弱序

C++map的key是const类型，不能修改，set的key非const可以修改，但不能修改用于排序的数据成员

遍历dict：

d={"zhang":60,"wang":70,"li":80};

print(d["zhang"]);

dict内部存放的顺序和key放入的顺序是没有关系的

默认情况下，dict迭代的是key。如果要迭代value，可以用for value in d.values()，如果要同时迭代key和value，可以用for k, v in d.items()

for v,k in d.items():

    print(v,":",k)

tp=d.items()

print(*type*(tp)) #<class 'dict\_items'>

d={'zcj',21,'lxy',20} #遍历dict的key

for v in d:

    print(v)

创建元素值都相同的list：

x=[True]\*5

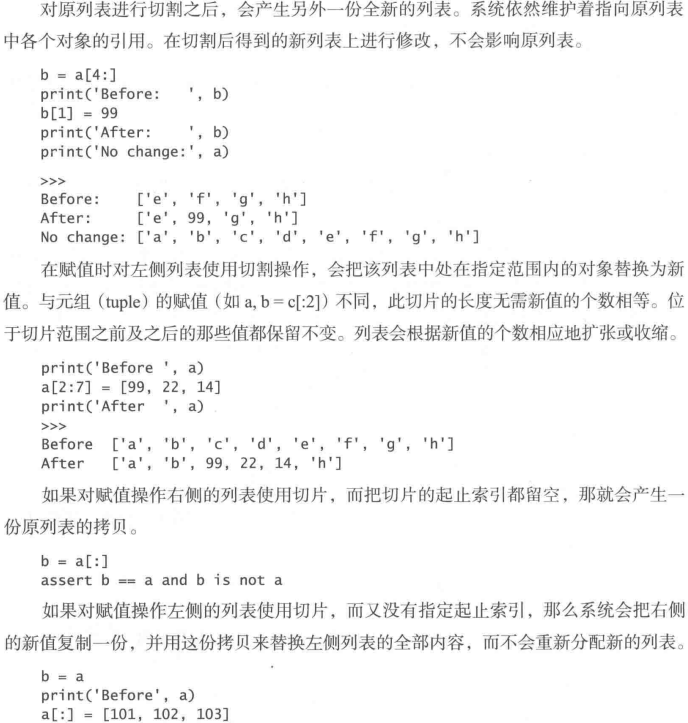
print(x); *#[True, True, True, True, True]*

d=[0]\*5 *#[0, 0, 0, 0, 0]*

切片操作：

L=list(range(101))  
print(L[10:31:3]) #10号元素取到31号元素，每隔3个取一个

对list切片，切片是原数据的拷贝



对列表切片时，即使start或end索引越界也不会出现问题：

inlist=[0,1,2,3,4]

first\_10\_eles=inlist[0:10]

last\_10\_eles=inlist[-10:]

print(first\_10\_eles)  #[0, 1, 2, 3, 4]

print(last\_10\_eles)   #[0, 1, 2, 3, 4]

字符串也可以看成是一种list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串：

name='zcjlxycxz'

ch=name[0:3]

print(ch)  #zcj

对ndarray切片，切片仍指向原数据

a=*list*(range(1,10)); #[1,2,3,4,5,6,7,8,9]

b=a[3:8:2]

b[1]=9

print(a) #[1, 2, 3, 4, 5, 6, 7, 8, 9]

切片操作的步长参数：

Python 中有一种常见的技巧，能够把字符串反转过来，这个技巧就是采用-1做步进值：

x='多谢'

y=x[::-1]

print(y) #谢多

这种技巧对对已经编码成UTF-8字节串的Unicode字符来说，则无法奏效：

w='谢谢'

x=w.encode('utf-8')

print(*type*(x))  #<class 'bytes'>

y=x[::-1]

z=y.decode('utf-8') #UnicodeDecodeError: 'utf-8' codec can't decode byte 0xa2 in position 0: invalid start byte

列表推导式：

a = [1, 2, 3, 4, 5]

squares = [x\*\*2 for x in a]

print(squares) #[1, 4, 9, 16, 25]

even\_squares = [x\*\*2 for x in a if x % 2 == 0]

print(even\_squares) #[4, 16]

dict、set有类似的推导机制：

score\_ranks = {'ghost':70, 'habanero':90, 'cayen':85}

cp={x for x in score\_ranks.items()} #{('cayen', 3), ('ghost', 1), ('habanero', 2)}

high\_scores={name:score for name,score in score\_ranks.items() if score>80}

print(high\_scores) #{'habanero': 90, 'cayen': 85}

name\_len\_set = {len(x) for x in score\_ranks.keys()} #{8, 5}

name\_len\_list = [len(x) for x in score\_ranks.keys()] #[5, 8, 5]

字符串型list转化为int型list：

x=[**'12'**,**'13'**,**'23'**,**'8'**]

x=[int(a) for a in x]

可迭代对象：

判断一个对象是不是可迭代对象：

**from** collections **import** Iterable

print(isinstance([1,2,34,5],Iterable)); #true

定义了\_\_iter\_\_()成员函数是可迭代对象的充分必要条件

from collections import Iterable

class A:

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

pass

print(isinstance(A(), Iterable)) #True

能用于for…in…循环的是可迭代对象

对于python内置的可迭代对象，可以使用函数iter()获取相应的迭代器（注意这个iter()是内置函数，与\_\_iter()\_\_不同)

迭代器定义了\_\_iter\_\_()和\_\_next()\_\_成员函数

实际上，iter(obj)是直接调用了obj的\_\_iter()\_\_方法，并且把这个方法的返回值作为自己的返回值。

在for..in…循环中,比如 for i in *list*,会先调用iter(*list*)获取可迭代对象list的迭代器，再每次调用迭代器的next()方法获取值。

s=[1,2,3,4]

for i in s.\_\_iter\_\_():

    print(i,*end*=' ') #1 2 3 4

for i in iter(s):

    print(i,*end*=' ') #1 2 3 4

for i in s:

    print(i,*end*=' ') #1 2 3 4

列表遍历：

**enumerate()**

* enumerate()是python的内置函数
* 对于一个可迭代的（iterable）对象（如列表、字符串），enumerate将其组成一个索引序列，利用它可以同时获得索引和值
* enumerate多用于在for循环中得到计数

test=np.array([[1,2,3],[10,20,30],[100,200,300]])

**for** i,s **in** enumerate(test):

print(i,**' '**,s)

0 [1 2 3]

1 [10 20 30]

2 [100 200 300]

**zip同时遍历多个可迭代对象：**

python3中zip将多个可迭代对象封装为一个生成器，python2是拼接为一个新的tuple

x=[1,2,3]

y=[10,11,12]

for i,j in zip(x,y):

print(i+j,end=' ') #11 13 15

name=['z','w','l','z']

icon=['@','&','$','%']

age=[21,23,20,18]

for x,y,z in zip(name,icon,age):

print(x,' ',y,' ',z)

#输出：

z @ 21

w & 23

l $ 20

z % 18

返回元素在列表中的索引：

a=['4','a','5','b','5','3']

print(a.index('5')) #2

#print(a.index('1')) #程序崩溃

python函数式编程

* 函数参数可以是别的函数名
* 函数名可以赋给变量；
* 函数名自身是变量，指向其它对象

>>> abs = 10

>>> abs(-10) # TypeError: 'int' object is not callable

把abs指向10后，就无法通过abs(-10)调用该函数了！因为abs这个变量已经不指向求绝对值函数而是指向一个整数10！

>>> a = abs *# 变量a指向abs函数*

>>> a(-1) *# 所以也可以通过a调用abs函数*

python函数返回多个值

python的函数可以返回多个值，实际上是返回了一个tuple()

print('abcd')

*def* func():

    return 'lxy',23

name,\_=func() #不能省略\_

print(name) #lxy

print(\_) #23

d=(3)

print(*type*(3)) #<class 'int'>

d=(3,)

print(*type*(d)) #<class 'tuple'>

print(len(d)) #1

#返回只包含一个元素的tuple

*def* func2():

    a=(30,) #不能写成(3)

    return a

a,=func2() #a是int

print(a) #3

b=func2() #b是tuple

print(b) #(3,)

函数默认参数：

python函数可以使用默认参数，默认参数在必选参数后、最好是不可变对象

如果默认参数是可变对象，则行为特殊：

*def* show(*a*,*b*=[1,2,3]):

    b+=[5,6]

    print(b)

show(0) #[1, 2, 3, 5, 6]

show(0) #[1, 2, 3, 5, 6, 5, 6]

# Python函数在定义的时候，默认参数的值就被计算出来了，下次调用时，还使用同一变量

# 如果默认参数是可变对象，则两次调用时值可能发生改变

函数星号参数*\*values*：

星号参数表示可选参数，实质是个tuple，不传参数时，tuple 0个元素，示例：

*def* log(*message*, \**values*): # The only difference

    if not values:

        print(message)

    else:

        va1ues\_str = ', '.join(*str*(x) for x in values)

        print(message,': ',va1ues\_str)

log('debug')  #debug

log('info',1,3,5)  #info :  1, 3, 5

code=[-1,-3,-5,-7]

log('warn',code) #warn :  [-1, -3, -5, -7]

log('warn',\*code) #warn :  -1, -3, -5, -7

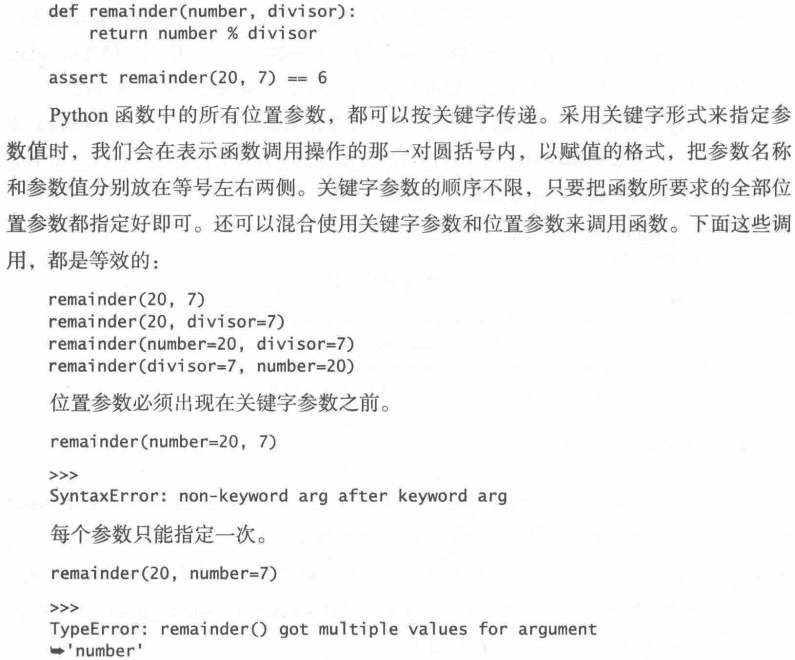
总结：

在def语句中使用\*args，即可令函数接受数量可变的位置参数。

调用函数时，可以采用\*操作符，把序列中的元素当成位置参数，传给该函数。

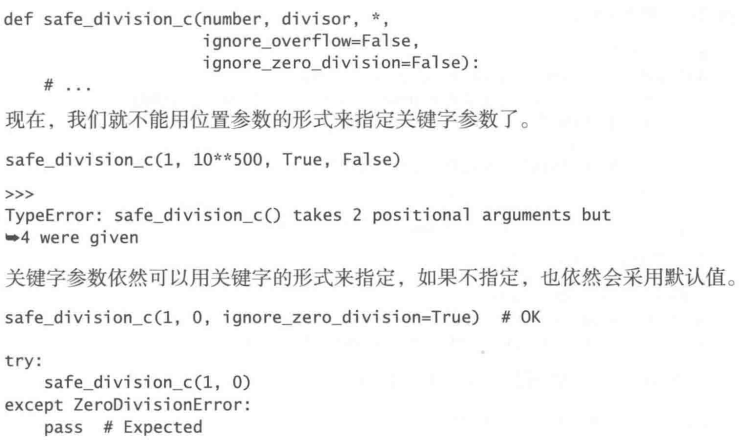
对生成器使用\*操作符，可能导致程序耗尽内存并崩溃

函数关键字参数：



只能以关键字形式指定参数：

参数列表里的\*号，标志着位置参数就此终结，之后的那些参数，都只能以关键字形式来指定



函数定义\调用时的星号（\*）

* 函数定义时使用单个\*会将所有的参数，放入一个元组（tuple）供函数使用。
* 使用两个\*\*会将所有的关键字参数，放入一个字典（dict）供函数使用。
* 函数调用时
* 在list、tuple、set前加一个星号会把容器中的所有元素解包（unpack）变成位置参数。
* 在dict前加一个星号会把字典的键变成位置参数。
* 在dict前加两个星号会把字典的键值对变成关键字参数。

*def* show(*name*,*age*,\*\**others*):

    print("name:",name,"age:",age,others)

show('lxy',21,*city*='changan',*score*='99')

闭包：

python 支持闭包（closure): 闭包是一种定义在某个作用域中的函数，这种函数引用了那个作用域里面的变量。

可变对象和不可变对象：

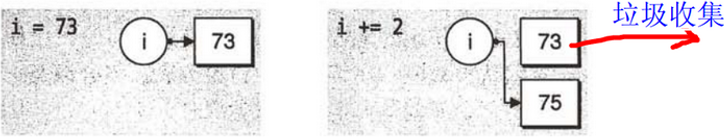
**不可变（immutable）对象：int、float、bool、string、元组（tuple)**

**可变（mutable）对象：list、dict、set**

不可变类型特点：

i = 73

i += 2



从上图可知，不可变对象的特征是对象自身没有变，只是创建了新对象，改变了变量的对象引用。

看一个例子：

>>>x = 1

>>>y = 1

>>> x is y

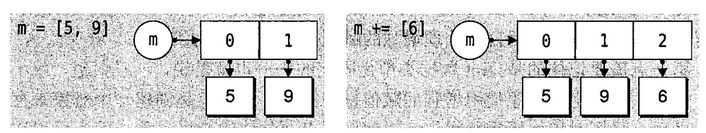
True

如上所示，因为整数为不可变，x,y在内存中均指向同一个内存地址，值得注意的是，整形来说，目前仅支持(-1,100)。

可变对象示例：

m=[5,9]

m+=[6]



函数传参时可变对象和不可变对象的区别：

test\_list = [1, 2, 3, 4]

test\_str = 'HAHA'

*def* func1(*alist*):

    blist=alist

    blist[2]=10

*def* func2(*astr*):

    bstr=astr

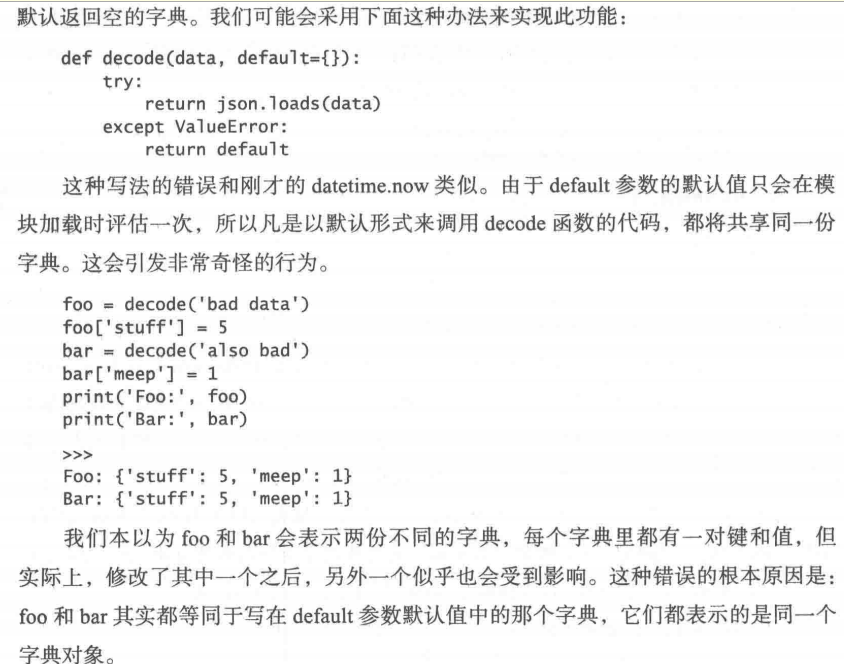
    bstr='ok'

func1(test\_list)

print(test\_list)  # [1, 2, 10, 4]改变了原来的值

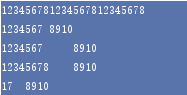
func2(test\_str)

print(test\_str)  # HAHA,没有变，也没法变



水平制表符’\t’：

print('123456781234567812345678')  
print('1234567\t8910')  
print('1234567\t\t8910')  
print('12345678\t8910')  
print('17\t8910')



类变量、成员变量、成员方法、类方法、staticmethod：

*class* A(*object*):

    cnt=0  #类变量，相当于C++中类的static成员变量

*def* \_\_init\_\_(*self*): #类的构造函数

        self.score=60  #类成员变量

        print('')

*def* improve\_score(*self*): #类成员方法

        old=self.score

        self.score+=10

        print('score from {} to {}'.format(old,self.score))

    @*classmethod*

*def* class\_func(*cls*): #类方法，相当于C++中类的static成员方法

        cls.cnt+=1

        print('cnt=',cls.cnt)

    @*staticmethod*

*def* static\_func():   #static method，仅用于封装

        #cnt+=1 #无法访问类变量、类成员变量

        print('hello')

x = A()

x.improve\_score()  #score from 60 to 70

x.class\_func()  #1

A.class\_func()  #2

A.static\_func() #hello

如果想要查看一个类是不是另一个类的子类，可以使用内建的 issubclass函数

如果想要检查一个对象是不是一个类的实例，可以使用内建的isinstance函数

类构造函数\_\_init\_\_：

python用内置方法\_\_init\_\_绑定数据成员。

python类的成员函数定义时第一个参数必须为self

python创建类的实例、通过实例调用成员函数时，省略self

\_\_init\_\_方法的第一参数永远是self，表示创建的类实例本身，因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。

*class* Student(*object*):

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

        self.nam = name

        self.scor = score

*def* show(*self*):

        print('student named ',self.nam,', score is',self.scor)

stu=Student('zcj',99)

print(stu.nam) #zcj

print(stu.scor) #99

stu.show() #student named zcj , score is 99

super().\_\_init\_\_初始化父类

*class* Base:

*def* \_\_init\_\_(*self*,*x*):

        self.val=x

*class* Son(*Base*):

*def* \_\_init\_\_(*self*,*x*):

*super*().\_\_init\_\_(x)

s=Son(8)

print(s.val)

类私有成员：

python中以双下划线\_\_开头的成员变量或方法，是private字段

python会对private成员的名称做一些简单的变换，以保证private成员的私密性。

当编译器看到self.\_\_score时，它会把\_\_score变换为\_A\_\_score，所以实际上无法访问私有属性，只不过是因为变换后的属性名与待访问的属性名不相符而已。

了解这套机制之后，就可以任意访问类的私有属性了，即，使用私有属性变换后的真实名称

*class* A():

*def* \_\_init\_\_(*self*):

        self.name='z'

        self.\_\_score=70

*def* \_\_show(*self*):

        print('name is:',self.name)

x = A()

print(x.name)

#print(x.\_\_score)  #AttributeError: 'A' object has no attribute '\_\_score'

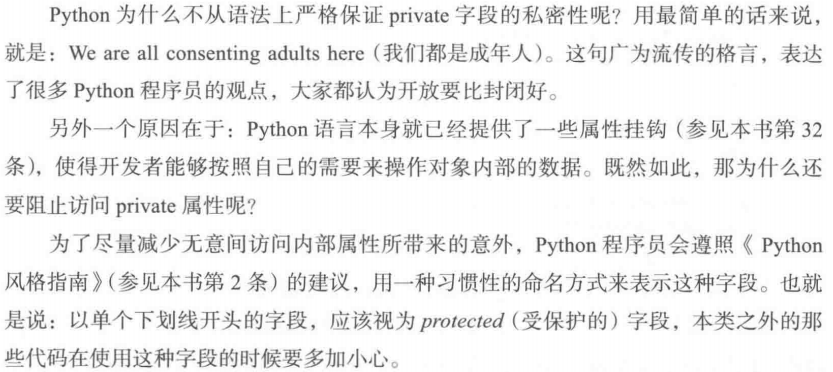
print(x.\_A\_\_score) #70

print(x.\_\_dict\_\_) #查看类实例的成员变量字典{'name': 'z', '\_A\_\_score': 70}

print(x.\_\_dir\_\_()) #查看类实例的所有成员变量及方法['name', '\_A\_\_score', '\_\_module\_\_', '\_\_init\_\_', '\_A\_\_show', '\_\_dict\_\_', '\_\_weakref\_\_', ……]

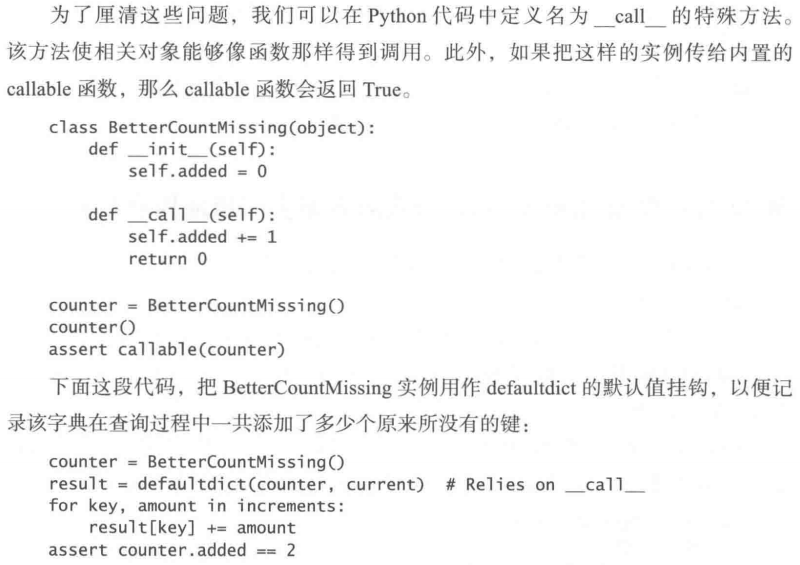
#x.show() #AttributeError: 'A' object has no attribute 'show'

x.\_A\_\_show() #z



类的\_\_call\_\_成员函数：

python中类的\_\_call\_\_函数类似C++中类的()运算符函数



defaultdict的作用是在于，当字典里的key不存在但被访问时，返回的不是keyError而是一个默认值，这个默认值由defaultdict的参数指定

dict =defaultdict( factory\_function)

这个factory\_function可以是list、set、str等等，作用是当key不存在时，返回的是工厂函数的默认值（即调用factory\_function()的返回值），比如list对应[ ]，str对应的是空字符串，int对应0，如下举例：

from collections import defaultdict

dict1 = defaultdict(int)

dict2 = defaultdict(str)

dict3 = defaultdict(set)

dict4 = defaultdict(list)

print(*int*())  #0

print(*list*()) #[]

print(*bool*()) #False

print(*dict*()) #{}

print(*set*()) #set{}

python的多线程不能真正多核并发：

CPython由C编写，是标准Python实现，也是其他Python编译器的参考实现。

CPython将Python源码编译成CPython字节码，然后由解释器解释执行这些字节码

CPython中有全局解释器锁（GIL）：同一时刻只有一条线程在执行python字节码

GIL虽然使得Python代码无法并行，但它对系统调用却没有任何负面影响。因为Python线程在执行系统调用的时候会释放GIL并且一直要等到执行完毕才会重新获取它

因此，python多线程对计算密集型任务无用，但对IO密集型任务有用

用C语言把程序中对性能要求较高的那部分代码，改写为扩展模块。由于C语言更贴近硬件，所以运行得比Python快，一旦运行速度达到要求，我们自然就不用再考虑平行计算了。此外C语言扩展也可以启动并平行地运行多条原生线程（nativethread)从而充分利用CPU的多个内核。

为了保证所有的线程都能够公平地执行，Python解释器会给每个线程分配大致相等的处理器时间。而为了达成这样的分配策略，Python系统可能当某个线程正在执行的时候，将其暂停（suspend)，后使另外一个线程继续往下执行

多线程threading：

import time

import threading

*class* A:

*def* \_\_init\_\_(*self*):

        self.running=False

        self.cap\_thr=None

*def* start(*self*):

        self.running=True

        self.cap\_thr=threading.Thread(*target*=self.\_\_doCapture,*args*=[2])

        self.cap\_thr.start()

*def* stop(*self*):

        if(self.running==True):

            self.running=False

            self.cap\_thr.join()

*def* \_\_doCapture(*self*,*interval*):

        while(self.running):

            print('echo')

            time.sleep(interval)

if \_\_name\_\_ == "\_\_main\_\_":

    a=A()

    a.start()

    time.sleep(10)

    a.stop()

python并行方式比较：

import time

import threading

from concurrent.futures import \*

*def* factorize(*number*) :

    for i in range(1,number + 1):

        if number % i == 0:

            yield i

*class* Worker:

*def* \_\_init\_\_(*self*):

        self.total\_cnt=0

*def* \_\_call\_\_(*self*,*x*):

        lst=*list*(factorize(x))

        cur\_cnt=len(lst)

        self.total\_cnt+=cur\_cnt

        print("x={}, factor cnt={}, total cnt={}".format(x,cur\_cnt,self.total\_cnt))

*def* test(\*,*method*):

    numbers = [4139079, 4214759, 4516637, 4852285]

    if(method==0):

        start = time.time()

        worker=Worker()

        for number in numbers:

            worker(number)

        end = time.time()

        print('method 0 took {*:.3*} seconds, total {} factor'.format((end - start),worker.total\_cnt))

    elif(method==1):

        start = time.time()

        worker=Worker()

        threads=[]

        for num in numbers:

            thr=threading.Thread(*target*=worker,*args*=[num])

            thr.start()

            threads.append(thr)

        for thr in threads:

            thr.join()

        end = time.time()

        print('method 1 took {*:.3*} seconds, total {} factor'.format((end - start),worker.total\_cnt))

    elif(method==2):

        start = time.time()

        worker=Worker()

        pool = ThreadPoolExecutor(len(numbers))

        futures=[]

        for num in numbers:

            fut = pool.submit(worker, num)

            futures.append(fut)

        for fut in futures:

            result=fut.result()

        end = time.time()

        end = time.time()

        print('method 2 took {*:.3*} seconds, total {} factor'.format((end - start),worker.total\_cnt))

    elif(method==3):

        start = time.time()

        worker=Worker()

        pool = ProcessPoolExecutor(len(numbers))

        futures=[]

        for num in numbers:

            fut = pool.submit(worker, num)

            futures.append(fut)

        for fut in futures:

            result=fut.result()

        end = time.time()

        print('method 3 took {*:.3*} seconds, total {} factor'.format((end - start),worker.total\_cnt))

if \_\_name\_\_ == "\_\_main\_\_":

    methods=[0,1,2,3]

    for i in methods:

        test(*method*=i)

#运行结果：

x=4139079, factor cnt=24, total cnt=24

x=4214759, factor cnt=8, total cnt=32

x=4516637, factor cnt=4, total cnt=36

x=4852285, factor cnt=4, total cnt=40

method 0 took 2.21 seconds, total 40 factor

x=4516637, factor cnt=4, total cnt=4

x=4214759, factor cnt=8, total cnt=12

x=4139079, factor cnt=24, total cnt=36

x=4852285, factor cnt=4, total cnt=40

method 1 took 2.13 seconds, total 40 factor

x=4516637, factor cnt=4, total cnt=4

x=4139079, factor cnt=24, total cnt=28

x=4214759, factor cnt=8, total cnt=36

x=4852285, factor cnt=4, total cnt=40

method 2 took 1.92 seconds, total 40 factor

x=4139079, factor cnt=24, total cnt=24

x=4516637, factor cnt=4, total cnt=4

x=4214759, factor cnt=8, total cnt=8

x=4852285, factor cnt=4, total cnt=4

method 3 took 1.16 seconds, total 0 factor

ProcessPooIExecutor类利用由multiprocessing模块所提供的底层机制，来逐步完成下列操作：

1)对每一项输入数据进行序列化，将其变成二进制形式。

2)通过本地套接字（local socket)将序列化之后的数据从主解释器所在的进程，发送到子解释器所在的进程。

3)接下来，在子进程中，对二进制数据进行反序列化操作，将其还原为Python对象。

4)引入包含Worker.worker函数的那个Python模块。

5)各条子进程平行地针对各自的输人数据，来运行Worker.worker函数。

6)……

numpy数值类型

numpy支持比Python更多种类的数值类型，如下：

|  |  |
| --- | --- |
| **序号** | **数据类型及描述** |
| bool\_ | 存储为一个字节的布尔值（真或假） |
| int\_ | 默认整数，相当于 C 的long，通常为int32或int64 |
| intc | 相当于 C 的int，通常为int32或int64 |
| intp | 用于索引的整数，相当于 C 的size\_t，通常为int32或int64 |
| int8 | 字节（-128 ~ 127） |
| int16 | 16 位整数（-32768 ~ 32767） |
| int32 | 32 位整数（-2147483648 ~ 2147483647） |
| int64 | 64 位整数（-9223372036854775808 ~ 9223372036854775807） |
| uint8 | 8 位无符号整数（0 ~ 255） |
| uint16 | 16 位无符号整数（0 ~ 65535） |
| uint32 | 32 位无符号整数（0 ~ 4294967295） |
| uint64 | 64 位无符号整数（0 ~ 18446744073709551615） |
| float\_ | float64的简写 |
| float16 | 半精度浮点：符号位，5 位指数，10 位尾数 |
| float32 | 单精度浮点：符号位，8 位指数，23 位尾数 |
| float64 | 双精度浮点：符号位，11 位指数，52 位尾数 |
| complex\_ | complex128的简写 |
| complex64 | 复数，由两个 32 位浮点表示（实部和虚部） |
| complex128 | 复数，由两个 64 位浮点表示（实部和虚部） |

类型有缩写，比如’u4’就代表np.unit32

numpy的matrix和ndarray：

import numpy as np

x=[[1,2,3],[1.5,2.5,3.5]]    #x是list

mat=np.mat(x)

print(*type*(mat)) #<class 'numpy.matrix'>

print(mat.T)   #矩阵的转置，list没有转置操作（即使list是矩阵形式）

x=[[1,2,3],[1.5,2.5,3.5]]

a=np.array(x)

print(*type*(a)) #<class 'numpy.ndarray'>

print(a.shape) #(2, 3)

print(a.T)

打印结果都是：

[[1.  1.5]

 [2.  2.5]

 [3.  3.5]]

numpy生成随机分布：

numpy.random.randn(d0, d1, …, dn)是从标准正态分布中返回一个或多个样本值，返回的是一个ndarray，shape为(d0, d1, …, dn)

numpy.random.rand(d0, d1, …, dn)的随机样本位于[0, 1)中，均匀分布

numpy.random.randint(low, high, size, dtype) ：生成在半开半闭区间[low, high)上离散均匀分布的整数值

import numpy as np

a=np.random.rand(2,3)

print(a)

[[0.20041731 0.71739625 0.23723261]

 [0.79588406 0.76638841 0.95199246]]

r=np.random.randint(2,6,(3,3),*dtype*='int64')

print(r)

print(r.dtype) #int64

[[4 5 2]

[5 3 5]

[3 3 5]]

Two-by-four array of samples from N(3, 6.25)：2.5\*np.random.randn(2,4)+3

np.zeros(shape)返回a new array of given shape filled with zeros. float类型

zeros(**shape**, dtype=float, order='C')

**>>>** s = (2,2)

**>>>** np.zeros(s)

array([[ 0., 0.],

[ 0., 0.]])

python产生不重复的随机数：

可以使用random模块中的random.sample函数，其用法如下：

import random

random.sample(population,k)

函数从序列或集合population中返回一个长度为k的随机数列表，并且列表中的随机数元素之间是不重复的，如：

>>>a = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

>>>b = random.sample(a,5)

>>>b #[1,14, 18, 19, 16]

图像名有中文时，opencv读取、保存

1. file = 'C:/测试.jpg'
2. **import** cv2
3. **import** numpy as np
4. img = cv2.imdecode(np.fromfile(file, dtype=np.uint8), -1)
5. cv2.imencode('.jpg',img)[1].tofile('C:/测试1.jpg')#保存

txt文件内容有中文时：

with open(src\_landmark,encoding='UTF-8') as f:

lines=f.readlines()

指定文件编码为utf-8：#coding=utf-8

matplotlib绘图：

**matplotlib绘图显示中文：**

为了在图表中能够显示中文和负号等，需要下面一段设置：

import matplotlib.pyplot as plt

plt.rcParams['font.sans-serif']=['SimHei'] 用来正常显示中文标签

plt.rcParams['axes.unicode\_minus']=False #用来正常显示负号

x=np.arange(1,10,0.5)

y=[np.cos(i) for i in x] #或者y=cos(x)

plt.plot(x,y)

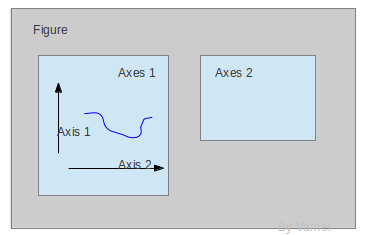
plt.show()

 plot(x, y, color='green', linestyle='dashed', marker='o',

             markerfacecolor='blue', markersize=12

**绘制子图subplot，设定坐标范围：**

* 如果你向plot()指令提供了一维的数组或列表，那么matplotlib将默认它是一系列的y值，并自动为你生成x的值。默认的x向量从0开始并且具有和y同样的长度
* 我们先来看什么是Figure和Axes对象。在matplotlib中，整个图像为一个Figure对象。在Figure对象中可以包含一个，或者多个Axes对象。每个Axes对象都是一个拥有自己坐标系统的绘图区域。



* plt.subplot(2,3,1)表示把图标分割成2\*3的网格。也可以简写plt.subplot(231)。其中，第一个参数是行数，第二个参数是列数，第三个参数表示图形的标号。
* plt.axis([xmin, xmax, ymin, ymax])，其中axis()命令给定了坐标范围。
* xlim(xmin, xmax)和ylim(ymin, ymax)来调整x,y坐标范围

**例：**

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**from** pylab **import** \* *#\*表示从pylab中导入所有东西*

mpl.rcParams[**'font.sans-serif'**] = [**'SimHei'**] *##用来正常显示中文*

matplotlib.rcParams[**'axes.unicode\_minus'**]=**False** *#用来正常显示负号*

x = np.arange(-5.0, 5.0, 0.02)

y1 = np.sin(x)

plt.figure(1)

plt.subplot(211)

plt.plot(x, y1)

plt.subplot(212)

*# #设置x轴范围*

*# xlim(-2.5, 2.5)*

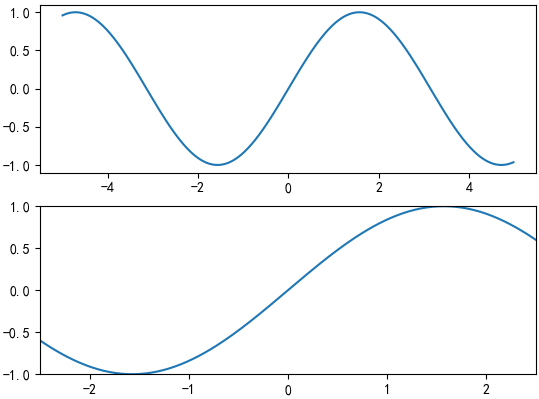
*# #设置y轴范围*

*# ylim(-1, 1)*

plt.axis([-2.5, 2.5, -1, 1])

plt.plot(x, y1)

plt.show()



**当前图、当前坐标，plot()和show()的关系：**

* 可以多次使用plt.figure()命令来产生多个图，图编号将按顺序自动增加，或者figure(10)主动赋值图编号为10。这里，要注意一个概念当前图和当前坐标。所有绘图操作仅对当前图和当前坐标有效
* figure()和subplot()
* 没有plt.figure()语句时，所有的更改都是在同一个figure上。
* plt.plot()、plt.title()、plt.xlabel()等操作引入的信息会累积保存，直到plt.plot()将所有信息一次性都显示在图形上。plt.show ()后，之前的所有信息被清空，再一次show()时图形中就不会包含这些信息。例如：如果先存入了一些信息A，然后plt.show()，再存入一些信息B，再plt.show()，运行，则先出现一幅图，显示信息A，关闭后，又出现一幅图，仅包含信息B。

**例1：**

**import** matplotlib.pyplot **as** plt

plt.figure(1) *# 第一张图*

plt.subplot(211) *# 第一张图中的第一张子图*

plt.plot([1,2,3,4,5])

plt.subplot(212) *# 第一张图中的第二张子图*

plt.plot([4,5,6,7,8])

plt.figure(2) *# 第二张图*

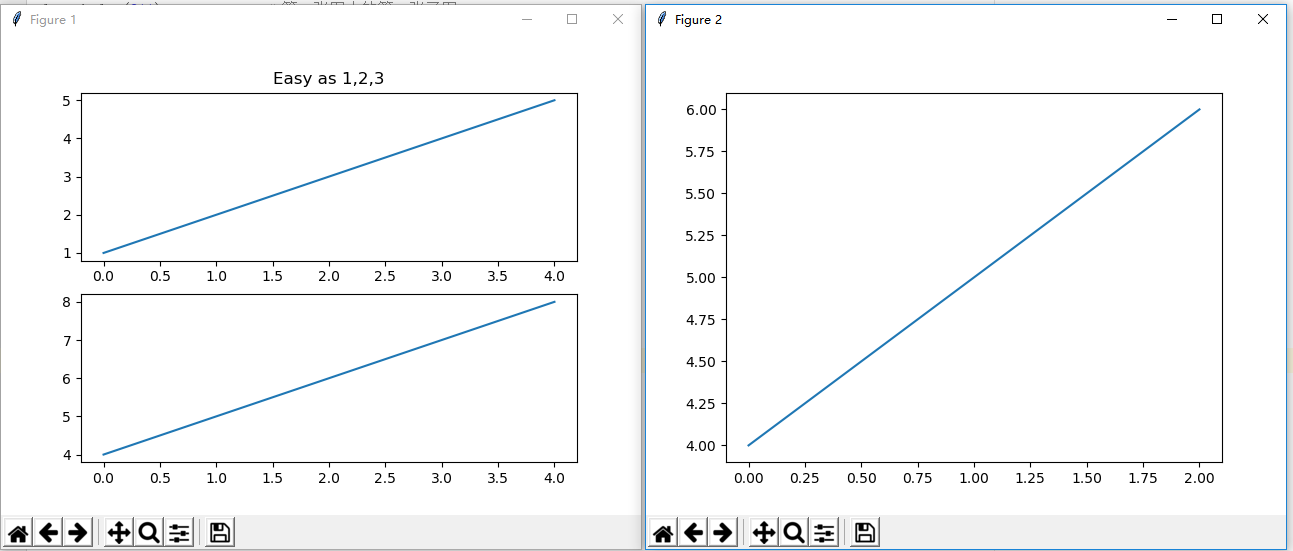
plt.plot([4,5,6]) *# 默认创建子图subplot(111)*

plt.figure(1) *# 切换到figure 1 ; 子图subplot(212)仍旧是当前图*

plt.subplot(211) *# 令子图subplot(211)成为figure1的当前图*

plt.title(**'Easy as 1,2,3'**) *# 添加subplot 211 的标题*

plt.show();



**例2：**

**import** matplotlib.pyplot **as** plt

plt.figure()

plt.plot(range(5));

plt.title(**u"name"**)

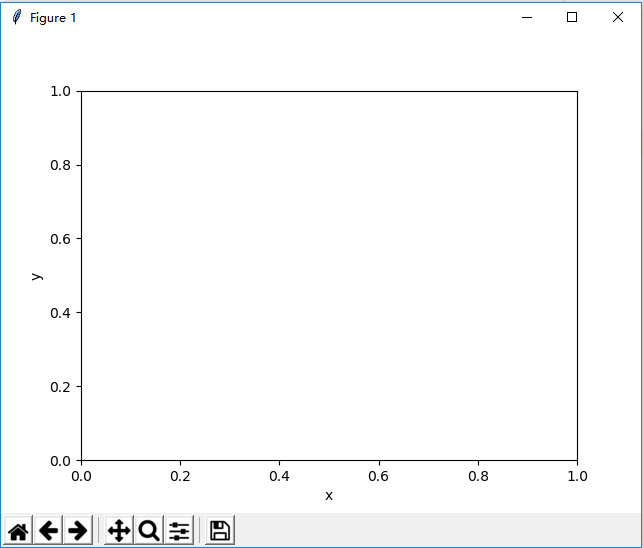
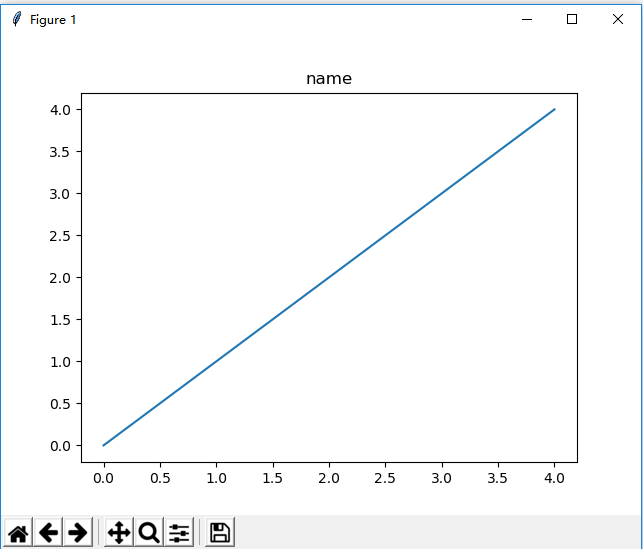
plt.show();

plt.xlabel(**u'x'**)

plt.ylabel(**u'y'**)

plt.show()

运行结果：先出现下方左边的图，关闭后，出现下右图



**plt.show(0)和plt.show()：**

**import** matplotlib.pyplot **as** plt

* plt.show(0)，即 plt.show(block=**False**)：matplotlib绘图显示的同时继续跑后边的代码； 如果plt.show(block=False)或plt.show(0)是程序最后一句代码时，则跑后边代码程序直接结束，而绘图还没显示完，看不到显示的图形
* plt.show()，即plt.show(block=**True**)：必须叉掉plot窗口，才会跑后边的代码。叉掉窗口后，相当于清除了当前的figure，如果后边的代码中又出现plt.plot()，则相当于新建了一个figure，再plot()，即相当于plt.figure()、plt.plot()

**例：**

**import** matplotlib.pyplot **as** plt

plt.figure()

plt.plot(range(5));

plt.title(**u"name"**)

plt.show(0)

**for** i **in** range(10000):

**for** j **in** range(10000):

j;

plt.xlabel(**u'x-axis'**)

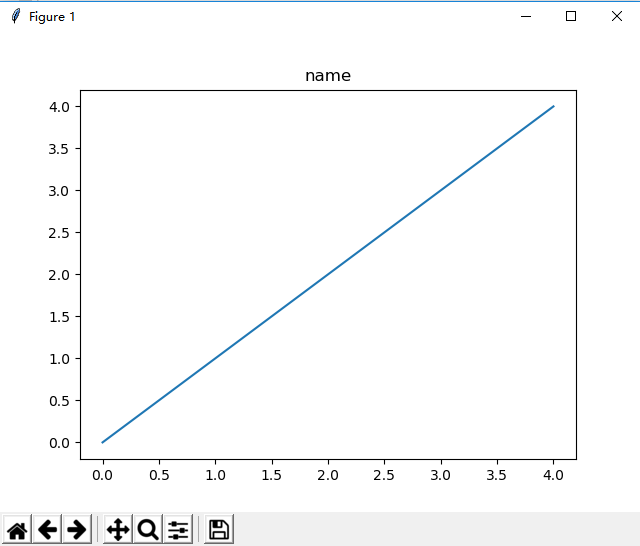
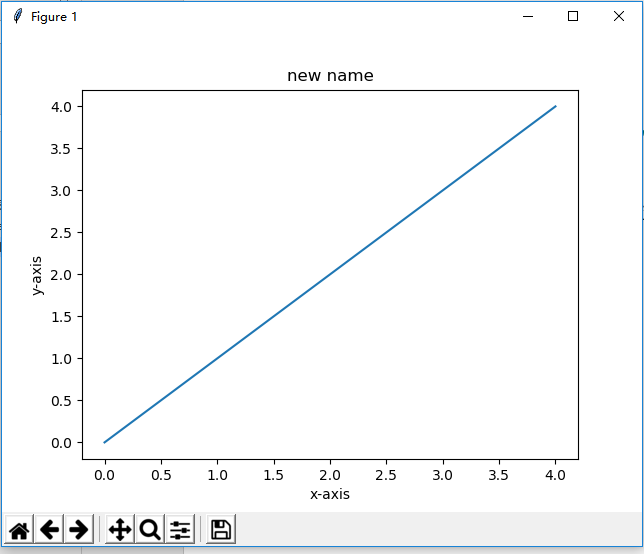
plt.ylabel(**u'y-axis'**)

plt.title(**u"new name"**)

plt.show()

运行结果：先出现下方左边的图，过一段时间，左边的图更改为下右图

第二次plot.show()前没有plot.figure()新建figure，也没有手动叉掉figure的操作，因此plt.title(**u"new name"**)在之前的figure上起作用，因而覆盖了之前的plt.title(**u"name"**)。

**清空plt绘制的内容**

plt.cla() #清除当前axes

plt.clf() #清除当前figure

plt.close(0)# 关闭图 0

plt.close('all') 关闭所有图

**matplotlib绘图的典型需求：**

* matplotlib显示图形一定时间后自动关闭

首先，不要使用plt.show() ,而改用为：

**plt.ion() #打开交互绘图功能**

**...plt.plot()…**

**plt.pause(15) #显示图形3秒**

**plt.close()**

**plt.ioff() #关闭交互模式**

**#**如果显示一定时间后只希望跑后边的代码而不想关闭图形，则去掉plt.close()语句

* matplotlib绘图的同时，跑后边的代码，且等待绘图完毕，整个程序才会结束 plt.show(0)
* 有多个figure需要绘制，且绘制除最后一个figure时希望代码都继续向后运行（即采用不阻塞方式），绘制出的各个figure一直存在，直至手动关闭

前几个figure采用plt.figure，plt.show(0)的方式，最后一个采用plt.figure，plt.show()

**例：**

plt.figure(0)

plt.plot(range(5))

plt.title(*u*'name')

plt.ion() #打开交互模式

plt.pause(3)

plt.close(0)

print([1,23,5,7])

plt.ioff() #关闭交互模式

*def* main():

    x=range(-50,51)

    for pow in range(1,5):

        y=[a\*\*pow for a in x]

        plt.figure(pow)

        plt.plot(y)

        if pow==4:

            plt.show()

        else:

            plt.show(0)

main()

运行结果：出现4幅图，叉掉每个figure后，打印’over’

numpy中 mean()函数功能：求取均值

经常操作的参数为axis，以m \* n矩阵举例：

axis 不设置值，对 m\*n 个数求均值，返回一个实数

axis = 0：压缩行，对各列求均值，返回 1\* n 矩阵

axis =1 ：压缩列，对各行求均值，返回 m \*1 矩阵

举例：

>>>  import numpy as np

>>> num1 = np.array([[1,2,3],[2,3,4],[3,4,5],[4,5,6]])

>>> now2 = np.mat(num1)

>>> now2

matrix([[1, 2, 3],

        [2, 3, 4],

        [3, 4, 5],

        [4, 5, 6]])

>>> np.mean(now2) # 对所有元素求均值

3.5

>>> np.mean(now2,0) # 压缩行，对各列求均值

matrix([[ 2.5,  3.5,  4.5]])

>>> np.mean(now2,1) # 压缩列，对各行求均值

matrix([[ 2.],

        [ 3.],

        [ 4.],

        [ 5.]])

**matrix和Numpy数组的乘法、加法**

1. 对于数组array乘就是对应位置的元素相乘，加就是对应位置的相加

X1 = np.array([[1,2], [3, 4]])

X2 = X1

print X2\*X1 #[[ 1 4] [ 9 16]]

print X2+X1 #[[2 4] [6 8]]

2. 对于矩阵matrix乘就是矩阵的点乘，加就是普通的矩阵相加，即对应位置相加：

X1 = np.matrix([[1,2], [3, 4]])

X2 = X1

print X2\*X1 #[[ 7 10] [15 22]]

print X2+X1 #[[2 4] [6 8]]

3.在numpy中存在很多的matrix和array 运算符混用的情况，程序也能通过，但这样很不好，尽量按照以上原则使用。如果2维的array想要进行矩阵的点乘运算，可以用np.dot(X1, X2)，如果matrix想要进行对应位置的乘，可以用np.multiply(X2,X1)

**np.mat()函数和np.matrix()函数的区别：**

Tmn = mat(vTmn)

如果vTmn是 matrix或者ndarray则此时Tmn和vTmn指向同一个对象

而 Tmn = matrix(vTmn) ，则Tmn是一个全新的对象

任何两个大小相等Numpy数组之间的运算，都是element-wise（点对点）

这种算数操作如果涉及标量（scalar）的话，会涉及到数组的每一个元素

a=np.array([[1/9,25,16],[1,1/36,4]],dtype=np.float64)

print(1/a\*\*0.5)

运行结果;

[[3. 0.2 0.25]

[1. 6. 0.5 ]]

两个数组比较会产生bool数组

a=np.array([[9,25,16],[1,36,4]])

b=np.array([[7,25,18],[3,70,1]])

comp=a>b

print('comp:\n',comp)

运行结果：

[[ True False False]

[False False True]]

**type()、.dtype、.shape函数查看矩阵或者数组的维数：**

type 获取数据类型，Python内置函数

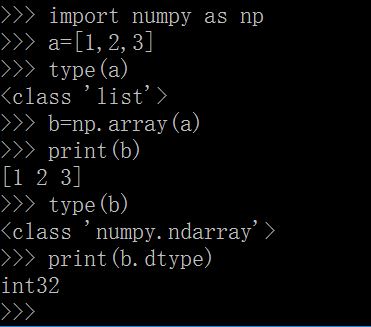
.dtype和.shape是Numpy数组对象的属性

2.dtype 数组元素的类型

.shape返回tuple类型 ()

type()返回类型<class 'type'>

.dtype返回类型<class 'numpy.dtype'>



建立一个4×2的矩阵c, c.shape[1] 为第一维的长度，c.shape[0] 为第二维的长度

1. >>> c = array([[1,1],[1,2],[1,3],[1,4]])
2. >>> c.shape
3. (4, 2)
4. >>> c.shape[0]
5. 4
6. >>> c.shape[1]
7. 2

**lamda表达式**

rotz=**lambda** theta:np.array([[np.cos(theta),-np.sin(theta),0],

[np.sin(theta),np.cos(theta),0],

[0,0,1]])

trans=np.array([2.12,-0.2,1.3]);

B=A.dot(rotz(np.pi/4).T)+trans

**矩阵的列交换次序**

A=np.random.randint(1,11,(2,3))

B=A[0:3,[2,1,0]]

print(A,'\n\n',B)

运行结果：

[[9 6 9]

[2 4 1]]

[[9 6 9]

[1 4 2]]

np.set\_printoptions(precision=3,suppress=**True**) #保留小数点后3位，不适用科学计数方法

一维array的长度：len(a)或者a.size

numpy.newaxis

插入新的维度 将shape为(3,)的array转化为(1,3)的二维array

>>> b = np.array([1, 2, 3, 4, 5, 6])

>>> b[np.newaxis]

array([[1, 2, 3, 4, 5, 6]])

>>> c = b[np.newaxis] ＃equals c = b[np.newaxis,:]

>>> b.shape

(6,)

>>> c.shape

(1, 6)

arr=[[1,2,3],[4,5,6]]

arr=np.array(arr)

print(arr)

print(arr.shape,'\nover\n')

x=arr[np.newaxis]

print(x)

print(x.shape)

[[1 2 3]

[4 5 6]]

(2, 3)

over

[[[1 2 3]

[4 5 6]]]

(1, 2, 3)

可迭代对象：以直接作用于for循环的对象统称为可迭代对象：Iterable

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

map()函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回

def f(x):

... return x \* x

...

>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> list(r)

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

map()传入的第一个参数是f，即函数对象本身。由于结果r是一个Iterator，Iterator是惰性序列（调用next()时才会计算下一个值），因此通过list()函数让它把整个序列都计算出来并返回一个list

将光标放在你想要复制的那行，然后Ctrl+d，就可以自动直接复制该行的功能，之后会在下一行显示出来

Series是一个定长的，有序的字典

Series可以像array一样直接参与一些数学运算

在许多应用中Series的一个重要功能是在算术用算中它会自动对齐

print(**"obj3\n"**,obj3,**'\n'**)

print(**"obj4\n"**,obj4,**'\n'**)

print(**'obj3+obj4\n'**,obj3+obj4,**'\n'**)

obj3

chang 200

jian 300

zhang 100

dtype: int64

obj4

he NaN

jun NaN

zhang 100.0

dtype: float64

obj3+obj4

chang NaN

he NaN

jian NaN

jun NaN

zhang 200.0

dtype: float64

print(**'dataframe start:'**)

data = {**'state'**: [**'Ohio'**, **'Ohio'**, **'Ohio'**, **'Nevada'**, **'Nevada'**],

**'year'**: [2000, 2001, 2002, 2001, 2002],

**'pop'**: [1.5, 1.7, 3.6, 2.4, 2.9]}

frame = DataFrame(data)

print(frame)

pop state year

0 1.5 Ohio 2000

1 1.7 Ohio 2001

2 3.6 Ohio 2002

3 2.4 Nevada 2001

4 2.9 Nevada 2002

给一个不存在的列赋值，将会创建一个新的列。 像字典一样 **del** 关键字将会删除列：

frame3[**'weight'**]=frame3.state #访问行frame3.ix[‘2nd’]

print(frame3)

**del** frame3[**'year'**]

print(frame3)

打乱次序

**>>>** arr = np.arange(10)

**>>>** np.random.shuffle(arr)

**>>>** arr

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

**tensorflow中one\_hot编码**

num\_classes=tf.constant(10) #10个类别

pred=tf.constant([9,0,1,3,5,7]) #6个样本，分别属于类别9、0、1……

onehot=tf.one\_hot(pred, num\_classes)

with tf.Session() as sess:

print(sess.run(onehot))

[[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]

[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]

I 表示int，3s表示三个字符长度的字符串，f 表示 float

损失函数loss与评价函数metrics：

* 评价函数和 [损失函数](https://keras.io/zh/losses) 相似，只不过评价函数的结果不会用于训练过程中。模型编译时（compile），评价函数应该作为 metrics 的参数来输入
* metrcis指定在训练模型过程中除了损失函数值之外的特定指标。一般对于分类问题来说，最常收集的指标就是准确率。需要收集的指标由设定数组中的名称决定

model.compile(loss='mean\_squared\_error',

optimizer='sgd',

metrics=[metrics.mae, metrics.categorical\_accuracy])

* 训练网络model.fit()之后，会返回一个历史对象（History object），其中包括了模型在训练中各项性能的摘要（包括每轮的损失函数值及在编译时制定收集的特定指标metrics）
* 使用model.compile(loss='mean\_squared\_error',optimizer='adagrad')时，训练过程model.fit(train\_x\_last,train\_y\_last,validation\_data=(val\_x\_last,val\_y\_last),epochs=200,batch\_size=64)中，每个epoch显示的是：



使用model.compile(loss='mean\_squared\_error',optimizer='adagrad',metrics=['mae'])时，训练过程model.fit(train\_x\_last,train\_y\_last,validation\_data=(val\_x\_last,val\_y\_last),epochs=200,batch\_size=64)中，每个epoch显示的是：



np.savetxt(**'wrist3DGL\_fromCNN.txt'**,wristGlobalLeft)

r=np.loadtxt(**'wrist3DGL\_fromCNN.txt'**)

其中，wristGlobalLeft是二维ndarray

C:\Users\myth>python C:\Users\myth\Desktop\a.py

桌面上有一个a.py文件，运行方法

Keras并不处理如张量乘法、卷积等底层操作。这些操作依赖于某种特定的、优化良好的张量操作库。Keras依赖于处理张量的库就称为“后端引擎”。Keras提供了三种后端引擎Theano/Tensorflow/CNTK，并将其函数统一封装，使得用户可以以同一个接口调用不同后端引擎的函数

可通过keras.backend.image\_data\_format()来获取当前的维度顺序。对2D数据来说，"channels\_last"假定维度顺序为(rows,cols,channels)而"channels\_first"假定维度顺序为(channels, rows, cols)。对3D数据而言，"channels\_last"假定(conv\_dim1, conv\_dim2, conv\_dim3, channels)，"channels\_first"则是(channels, conv\_dim1, conv\_dim2, conv\_dim3)

isinstance(x, np.ndarray):

x=np.array([1,2,3])

print(x.dtype) #int32

y=np.array([1,2.8,3])

print(y.dtype) #float64

st='abcdefghigk'

print(st[:3])

print(st[:-3])

print(st[6:])

print(st[-6:])

print(st[-6:-10])#空

print(st[2:8:3])#cf

print(st[2:8:-1])#空

print(st[2:8:-3]) #空

print(st[:5:-3]) #k h

print(st[-2::-1]) #gihgfedcba

print(st[-10:-6:-2])#空

str[i,j,k]

没有k，则相当于默认k为1

从偏移为i直到偏移为j-1，每隔k元素索引一次

如果k>0，则要求str[j]在str[i]右边，否则为空

如果k<0，则要求str[j]在str[i]左边，否则为空

k>0，从左往右

k<0，从右往左

Python 模块(Module)，是一个 Python 文件，以 .py 结尾，包含了 Python 对象定义和Python语句

ensorflow使用五级日志：调试DEBUG，信息INFO，警告WARN，错误ERROR和致命FATAL

默认情况下，TensorFlow配置在日志记录级别的WARN，但当跟踪模型的训练，你会想要调整水平到INFO

tf.logging.set\_verbosity(tf.logging.INFO)

slim库是tensorflow中的一个高层封装，它将原来很多tf中复杂的函数进一步封装，省去了很多重复的参数，以及平时不会考虑到的参数

**维度与axis：**

维度是用来索引一个多维数组中某个具体数所需要最少的坐标数量

再多的维只不过是是把上一个维度当作自己的元素

1维，又称1维张量，数组，vector：[1, 2, 3]

2维，又称2维张量，矩阵，二维数组：[[1,2], [3,4]]

3维，又称3维张量，立方（cube），三维数组：[ [[1,2], [3,4]], [[5,6], [7,8]] ]

tf.abs(a)对a中每个元素取绝对值，输出形状不变

temp=np.random.rand(2,3,5,4)

a=(temp\*10).astype('int32')

print(a)

print(np.mean(a,keepdims=False,axis=0))，axis=0则输出3×5×4，axis=1则输出2×5×4，axis=2则输出2×3×4，axis=3则输出2×3×5

print(np.mean(a,keepdims=False,axis=(0,1)))，axis=(0,1)则输出5×4，axis=(1,3)则输出2×5

如果keepdims=True，则输出的维度数不变，相应位置的为1，即：

print(np.mean(a,keepdims=True,axis=(0,1)))，axis=(0,1)则输出1×1×5×4，axis=(1,3)则输出2×1×5×1

**多维array切片存取：**

id4=np.identity(4)

t=(np.array([10,20,30])).T

r=np.array([[1,1,1],[2,2,2],[3,3,3]]);

id4[0:3,0:3]=r; #取0、1、2行，0、1、2列部分

id4[0:3,3]=t #取0、1、2行，3列部分

print(t,**'\n\n'**,r,**'\n\n'**,id4)

[10 20 30]

[[1 1 1]

[2 2 2]

[3 3 3]]

[[ 1. 1. 1. 10.]

[ 2. 2. 2. 20.]

[ 3. 3. 3. 30.]

[ 0. 0. 0. 1.]]

**在编写代码的时候，总是要先定义好整个图，然后才调用sess.run()**

**使用PIL库resize图像**

from PIL import Image #resize()

srcImage=Image.open(filepath)

image=srcImage.resize((batch\_shape[1], batch\_shape[2]))

image=np.array(image) #转为Numpy数组

**使用skimage库将[0,1]之间的float转为[0,255]之间的UINT**

from skimage import img\_as\_ubyte

dst=img\_as\_ubyte(src)

或者src是Numpy数组，则可以使用numpy的astype函数：dst=(src\*255.0).astype('uint8')

**OpenCV显示、保存图片：**

srcImage=cv2.imread(filepath) #返回类型为 <class 'numpy.ndarray'>

cv2.imshow("src",srcImage)

while(1):

if(cv2.waitKey(0)==13): #除非按回车键，否则一直阻塞，13即’\r’，即回车

break

13可以换成==ord('\r')，ord获取单字符的Unicode code point

**让任意程序暂停又恢复：**

while(1):

st=input()

if st=="":

break;

目前OpenCV的imwrite("cat1.png",dst)只支持单通道和3通道的图像，并且要求其深度为8bit和16bit无符号；如果dst是float，则转为UINT8再存：当dst是[0,1]之间的float，则存一副全黑的图；当dst是[0,255]之间的float，则float转int损失了一些精度再存

即使输出文件夹没有写入权限，Opencv也会从imwrite中安静地返回

**路径描述**

在windows读取文件可以用\，但在字符串里面\被作为转义字符使用，那么在描述路径时有两种方式

'c:\\a.txt'，转义的方式

r'c:\a.txt'，声明字符串不需要转义

os.path.join()返回字符串

os.getcwd()返回字符串

a=os.getcwd()+'\data.png' #a是字符串

等价于a= os.path.join(os.getcwd(),'dat')

path1=r"C:\Users\myth\Pictures\beauty"

path1=path1+'\car.png'

**glob模块：**

glob是python自己带的一个文件操作相关模块

import glob

names=glob.glob(r"C:\Users\myth\E\Dataset\一些动物图片\originals\\*.png") #定义了文件路径匹配规则,返回一个列表

\* 匹配0或多个字符

? 匹配1个字符

numpy.array()将嵌套list被转化为多维Numpy数组

w=np.array([[[1,4,6],[20,9,8]],[[4,4,4],[5,5,5]]])，w是2×2×3的Numpy数组

.ndim输出维度数

range() 函数返回的是一个可迭代对象，而不是一个list

arange是一个数组版的python range函数

a=range(3)

print('a:\n',a) # range(0, 3)

print('type(a):',type(a)) # <class 'range'>

b=np.arange(3)

print('b:\n',b) #[0 1 2]

print('type(b):',type(b)) #<class 'numpy.ndarray'>

astype()将字符串转为flaot

st=np.array(['12','20','50'])

print(st) #['12' '20' '50']

print(st.astype(np.float64)) # [12. 20. 50.]

可以用其他数组的dtype直接来制定类型

a=np.array([3,5,8])

b=np.array([1,3.0,5.2],dtype=np.float64)

c=a.astype(b.dtype)

print('c:\n',c)# [3. 5. 8.]

**list切片与Numpy数组切片的区别**

python内建的list与Numpy的array有个明显的区别，这里array的切片后的结果只是一个views（视图），用来代表原有array对应的元素，而不是创建了一个新的array。但list里的切片是产生了一个新的list

ar=np.array([0,1,2,3,4,5])

ar[2:-1]=[12,12,12] #ar变为[0,1,12,12,12,5]

pt=ar[2:-1] #产生视图

pt=[7,8,9] #给另一个对象起名为pt

print('ar:',ar) #[0,1,12,12,12,5]

ar=np.array([0,1,2,3,4,5])

ar[2:-1]=[12,12,12] #ar变为[0,1,12,12,12,5]

pt=ar[2:-1] #产生视图

pt[:]=[7,8,9] #改变了pt，而pt代表原有数组对应的元素，因此改变了ar

print('ar:',ar) #[0,1,7,8,9,5]

ls=[0,1,2,3,4,5]

ls[2:-1]=[12,12,12] #ls变为[0,1,12,12,12,5]

pt=ls[2:-1] #产生一个新的list

pt=[7,8,9] #另一个对象起名为pt

print('ls:',ls) #[0,1,12,12,12,5]

ls=[0,1,2,3,4,5]

ls[2:-1]=[12,12,12] #ls变为[0,1,12,12,12,5]

pt=ls[2:-1] #产生一个新的list

pt[:]=[7,8,9] #改变列表pt

print('ls:',ls) #[0,1,12,12,12,5]

如果想获得一个新的Numpy数组，copy()

w=np.array([[[1,4,6],[20,9,8]],[[4,4,4],[5,5,5]]]).astype(np.int8)

car=w[1].copy()

对于多维数组，如果省略后面的索引，返回的将是一个低纬度的多维数组

如w是2×2×3的Numpy数组，则w[1]是2×3的Numpy数组

w=np.array([[[1,4,6],[20,9,8]],[[4,4,4],[5,5,5]]]).astype(np.int8)

pt=w[1]

pt[:]=np.array([[10,11,12],[20,21,22]])

print('w:\n',w) #[[[1,4,6],[20,9,8]], [[10,11,12],[20,21,22]]]

标量和数组都能赋给Numpy数组的切片

x=np.random.randn(3,5)

x[:]=1

print(x) #x变为3×5的全1数组

**布尔索引**

用布尔索引总是会返回一份新创建的数据，原本的数据不会被改变src=np.random.randint(10,100,(4,2))

cond=np.empty(shape=src.shape[0],dtype=np.bool\_)

cond[:]=[True,False,True,False]

dst=src[cond]

print('src:\n',src)

print('dst:\n',dst)

运行结果：

src:

[[25 16]

[77 31]

[67 35]

[32 79]]

dst:

[[25 16]

[67 35]]

**花式索引按一定顺序选出几行**

可以用一个整数list或整数ndarray来指定顺序

花式索引返回一个新的ndarray

src=np.random.randint(10,100,(4,4))

dst=src[[3,1],:]

print('src:\n',src)

print('dst:\n',dst)

t=np.array([[1,2,3,4],[5,6,7,8]])

dst[:,:]=t

print('src:\n',src)

运行结果：

src:

[[95 53 28 11]

[76 61 55 43]

[11 26 31 84]

[93 86 70 53]]

dst:

[[93 86 70 53]

[76 61 55 43]]

src:

[[95 53 28 11]

[76 61 55 43]

[11 26 31 84]

[93 86 70 53]]

**花式索引选单独的几个元素**

arr = np.arange(32).reshape((8, 4))

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, 27],

[28, 29, 30, 31]])

arr[[1, 5, 7, 2], [0, 3, 1, 2]] #array([ 4, 23, 29, 10])

可以看到[ 4, 23, 29, 10]分别对应(1, 0), (5, 3), (7, 1), (2, 2)

**从ndarray中选长方形区域：**

arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]

先花式索引选行，再花式索引选列

先从arr中选出[1, 5, 7, 2]这四行，然后[:, [0, 3, 1, 2]]表示选中所有行，但是列的顺序要按0,3,1,2来排

**ndarray转置：**

转置也是返回一个view，而不是新建一个数组

e=np.arange(6).reshape(3,2)

print('e:\n',e) #[[0 1] [2 3] [4 5]]

x=e.T

print('x:\n',x) # [[0 2 4][1 3 5]]

x[0,:]=np.array([8,9,10]) #改变了x，因为x是原ndarray的一个view，因此改变了原ndarray

print('x:\n',x) # [[ 8 9 10][ 1 3 5]]

print('e:\n',e) #[[ 8 1] [ 9 3] [10 5]]

arr = np.arange(16).reshape((2, 2, 4))

arr.transpose((1, 0, 2)) #0号轴和1号轴交换

文件读写

调用read()会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用read(size)方法，每次最多读取size个字节的内容。另外，调用readline()可以每次读取一行内容，调用readlines()一次读取所有内容并按行返回list

with open('/path/to/file', 'r') as f:

print f.read()

**读写ndarray到txt文件**

a=np.arange(48).reshape(6,8)

print(a)

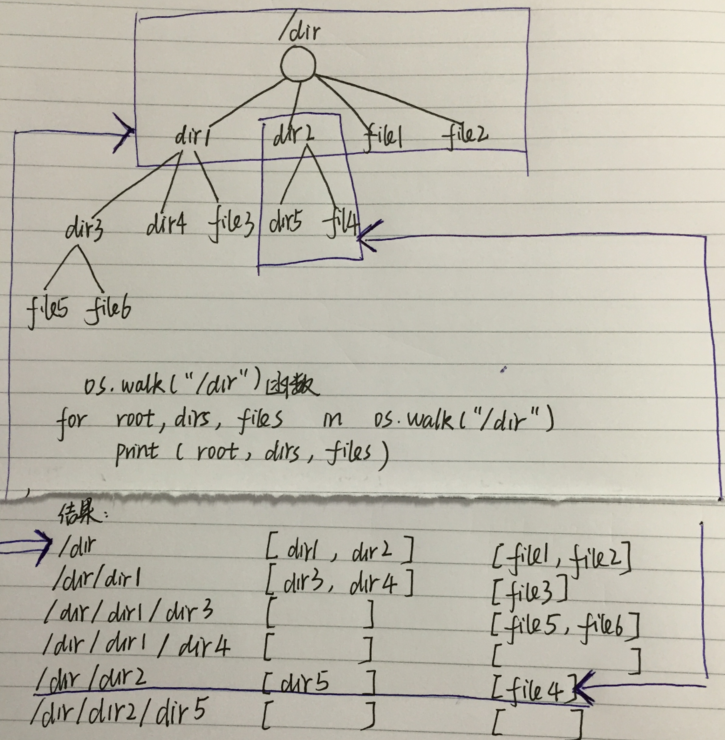
print(np.savetxt('arr.txt',a))

b=np.loadtxt('arr.txt')

print(b)

注意数组a的元素类型为int，但保存到txt中后变为float，再读取为b，b也是float

**os.walk()遍历文件夹：**



os.walk()返回一个<class 'generator'>

def Test1(rootDir):

list\_dirs = os.walk(rootDir)

print('\n',type(list\_dirs)) # <class 'generator'>

print(list(list\_dirs)) #将generator转为list，list的每个元素是一个3元tuple，tuple的第一个元素是目录名字符串，第二个元素是子目录(子文件夹)名字符串组成的list，第三个元素是文件名字符串组成的list

for (root, dirs, files) in os.walk(rootDir) : #如果os.walk(rootDir)换成list\_dirs，则不能打印出任何信息

for f in files:

print(f)

**shutil模块操作文件**

import shutil #文件操作工具

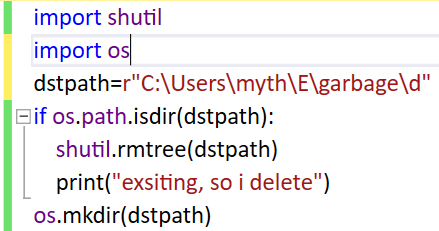
shutil.copy(src\_full\_name, dst\_full\_name)

判断字符串name中是否含有字串：if '0.1' in name

**创建删除目录：**

os.removedirs 要求空文件夹

shutil.rmtree(dirname) 无论是空文件夹还是非空文件夹都能删除，如果是非空文件夹，则删除该文件夹下所有内容（包括子文件夹、子文件夹的子文件夹）。如果目录不存在，则程序报错。



判断文件夹是否存在：os.path.isdir(dirname)

创建文件夹：os.mkdir(dirname),如果已存在，则程序报错

**这两个函数之间最大的区别是当父目录不存在的时候os.mkdir(path)报错不会创建，os.makedirs(path)则会创建父目录**

os.path.isdir(name)判断是否存在名为name的目录，os.path.exists(name)判断是否存在名为name的目录或文件

a是一个文件

dstpath=r"C:\Users\myth\E\garbage\a"

print(os.path.isdir(dstpath)) #False

print(os.path.exists(dstpath)) #Ture

os.removedirs(path)

递归地删除目录。类似于rmdir(), 如果子目录被成功删除， removedirs() 将会删除父目录；但子目录没有成功删除，将抛出错误

for dirPath, dirNames, fileNames in os.walk(scrPath):

for name in fileNames:

if '0.1' in name and random.randint(0,100) < 10:

src\_full\_name = os.path.join(dirPath, name)

dst\_full\_name = os.path.join(dstPath, name)

shutil.copy(src\_full\_name, dst\_full\_name)

**namedtuple 和 deque 和defaultdict**

#namedtuple是一个函数，它用来创建一个自定义的tuple对象，

#并且规定了tuple元素的个数，并可以用属性而不是索引来引用tuple的某个元素

from collections import namedtuple

pDepth=namedtuple('pointDepth',['row','col','depth'])

p1=pDepth(320,450,500)

print(p1) #pDepth(row=320, col=450, depth=500)

print(p1.depth) #500

print(isinstance(p1,tuple)) #True

#deque除了实现list的append()和pop()外，还支持appendleft()和popleft()，

#这样就可以非常高效地往头部添加或删除元素

from collections import deque

dp=deque([1,3,5,7,9])

dp.pop()

dp.append(11)

dp.popleft()

dp.appendleft(-1)

print(dp) # deque([-1, 3, 5, 7, 11])

dt={'zhang':25,'wang':23,'lv':21}

dt['wang']=28

print(dt)

print(dt['zhang']) #如果不存在对应的key，则报错，如print(dt['li'])

#key不存在时，返回一个默认值，就可以用defaultdict

from collections import defaultdict

dct={'zhang':225,'wang':223,'lv':221}

#dct=dict([('zhang',225),('wang',223),('lv',221)]) #与上一句等价

dd=defaultdict(lambda:'nonexsit',dct)

print(dd)

print(dd['li'])

print('issubclass(defaultdict,dict):',issubclass(defaultdict,dict)) #True

**numpy.newaxis：**

增加一个维度

b=np.array([1,2,3,4,5])

c=b[np.newaxis] #等价于c=b[None]

print(b.shape) #(5,)

print(c.shape) #(1,5)

print(c) #[[1 2 3 4 5]]

**numpy.linspace()生成等差数列：**

print(np.arange(10,2,-2,dtype=np.float64)) #arange 是通过设置样本之间的差值(可以为负)来生成数组的

print(np.linspace(10,2,4,False)) #linespace通过设置样本数目来生成数组

numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)

start：起始值 stop：结束值

num：生成的个数

endpoint True:包含 False：不包含 默认True

restep：显示相邻两数之差 默认不显示

dtype: 输出类型

**numpy.logspace()生成等比数列：**

x=np.logspace(0,2,10,True)

[ 1. 1.66810054 2.7825594 4.64158883 7.74263683

12.91549665 21.5443469 35.93813664 59.94842503 100. ]

在logspace中，起始位和终止位代表的是10的幂（默认基数为10），0代表10的0次方，2代表10的2次方. 如果不想包含尾节点，可以再加一个参数endpoint，默认 设置为1，当必要时，会适当调整base以适应个数的变化

**求平均数、中位数、众数：**

from scipy import stats

ls=[1,3,5,7,2,4,5,6,7,6,3,5,3,9]

print("众数：",stats.mode(ls)[0][0])

import numpy as np

print("mean: ",np.mean(ls))

print("中位数: ",np.median(ls))

**排序：**

用sort函数对列表排序时会影响列表本身，而sorted不会

sort()和sorted()默认由小到大，sorted()设置reverse参数为True可实现由大到小

argsort()返回由小到大排序后对应的索引，取负数再argsort()则是由大到小排序后对应的索引

age=np.array([3,5,1,10,5])

print(age.max()," ",age.argmax())#10 3

print(age.min()," ",age.argmin()) #1 2

smallTobig=sorted(age)

print(smallTobig) #[1, 3, 5, 5, 10]

print(smallTobig)#[1, 3, 5, 5, 10]

bs=sorted(age,reverse=True)

print(bs) #[10, 5, 5, 3, 1]

print(age) #[ 3 5 1 10 5]

print(np.argsort(age))#[2 0 1 4 3]

print(np.argsort(-age)) #[3 1 4 0 2]

print(age) #[ 3 5 1 10 5]

tmp=age.sort()

print(tmp) #None

print(age) #[ 1 3 5 5 10]

**列表和字符串相加（连接）：**

a="123"

b=a+"hello"

print(b) #123hello

a=[]

b=[1,3]

c=[2,6]

d=a+b+c

print(d) #[1, 3, 2, 6]

**os.walk()：**

srcImgName=[]

for dirPath, dirNames, fileNames in os.walk(src\_img\_dir):

for name in fileNames:

if name.split(".")[-1]=="jpg":

srcImgName.append(name)

name="hello\n"

print(name)

print("src")

new=name[0:-1]

print(new)

print("dst")

运行结果：

hello

src

hello

dst

请按任意键继续. . .

name=r"/data/zhangchangjian/eye\_mouth\_train/bgr/SR300\_1535006296391680\_frame\_177\_bgr.jpg"

temp=""

for i in name:

if i=="/":

temp+="\_"

else:

temp+=i

print(temp) #\_data\_zhangchangjian\_eye\_mouth\_train\_bgr\_SR300\_1535006296391680\_frame\_177\_bgr.jpg

temp=name.replace("/","\_")

print(temp)#\_data\_zhangchangjian\_eye\_mouth\_train\_bgr\_SR300\_1535006296391680\_frame\_177\_bgr.jpg

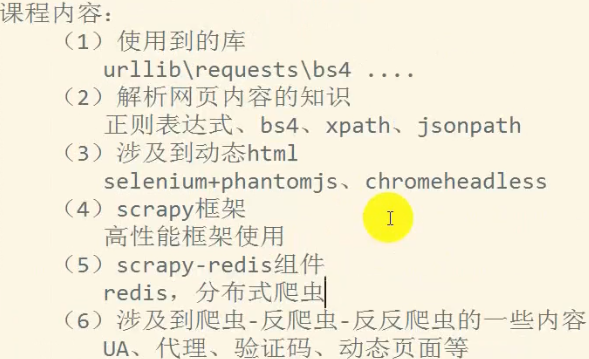
**爬虫：**

通用爬虫如google、baidu等如何抓取新网站？

* 主动提交url
* 让其它网站帮忙设置友情链接
* 百度和DNS服务商合作，抓取新网站

爬虫思路：

给一个url，模拟浏览器访问url，解析返回的数据



http：明文传输

安全套接字层超文本传输协议：https

加密、解密 用相同的密钥，则对称加解密

加密、解密 用不同的密钥，则非对称加解密

服务端给所有客户端公钥，仅自己知道私钥，客户用公钥加密数据发给服务端，服务端用私钥解密。



一个http请求包括：请求行、多个消息头、一个空行、实体内容



请求方式：GET、POST……。默认情况下浏览器向服务器发送的都是get请求，例如在浏览器直接输地址访问，点超链接访问等都是get

消息头：

accept:浏览器通过这个头告诉服务器，它所支持的数据类型

　　Accept-Charset: 浏览器通过这个头告诉服务器，它支持哪种字符集

　　Accept-Encoding：浏览器通过这个头告诉服务器，支持的压缩格式

Accept-Language：浏览器通过这个头告诉服务器，它的语言环境

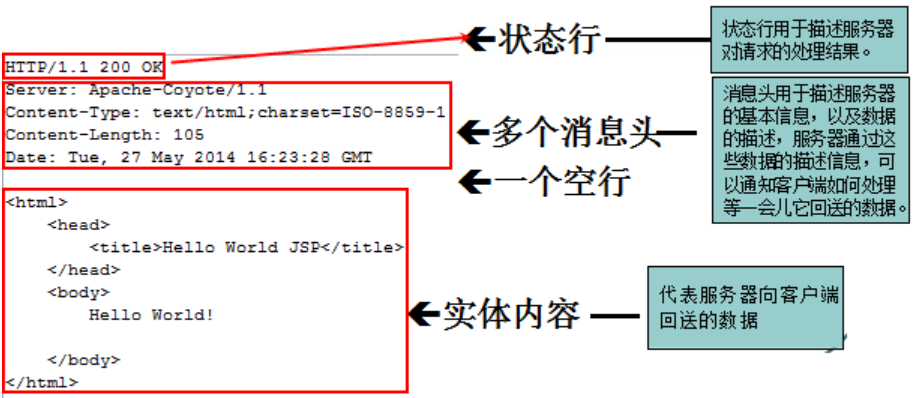
Host：浏览器通过这个头告诉服务器，想访问哪台主机

If-Modified-Since: 浏览器通过这个头告诉服务器，缓存数据的时间

Referer：浏览器通过这个头告诉服务器，客户机是哪个页面来的 防盗链

Connection：浏览器通过这个头告诉服务器，请求完后是断开链接还是何持链接

一个响应包括：状态行、多个消息头、一个空行、实体内容



状态码200： ok，www服务器提供了请求的网页

301 (永久移动)：请求的网页已永久移动到新位置。 www服务器返回此响应时，会自动将请求者转到新位置。以后客户可以使用新位置的url请求

302 (临时移动)： www服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求

403 (禁止)： www服务器拒绝请求。

404 (未找到)： www服务器找不到请求的网页

一个网页的呈现，中间不止一次http请求。平均一个网页大概10~15个http请求

谷歌浏览器抓包工具：网页右键->请求->右边栏的network，再查看每个请求的headers、responds、timing等

fiddler专业抓包软件：

< >：html内容、

{json}、json内容，很可能是个接口

{css}：css文件

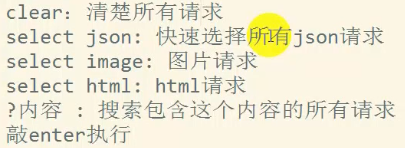
菜单file->capture traffic：停止抓取

叉号->remove all:删除已经抓取的所有内容

抓包结果，点击某个，点击右边的inspectors。右上方：点击Raw，显示请求的所有信息；WebForms显示GET和POST请求所带参数

右下：点击ImageView、WebView可视化显示；点击Raw看响应的所有信息；点击JSON查看json内容

左下黑色框：输入指令



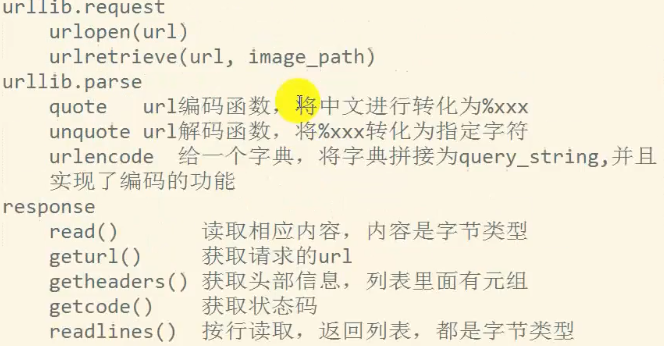
urllib库：Python自带，模拟浏览器发请求

Python2：urllib、urllib2两个库

Python3：只有urllib一个库，里有urllib.request、urllib.parse

显示每张图片都要单独发送请求

右键图片->复制图片地址



**json串中有特殊字符时：**

loads带strict=False参数

print的时候虽然打印不出来value，但是再转为byte串，这个特殊字符还在

import json

#同步空闲符

sync\_byte=*bytearray*(1)

sync\_byte[0]=22

#字节串{"a":"同步空闲符b"}

json\_bin=*bytearray*()

json\_bin+=(*r*'{"a":"').encode()

json\_bin+=sync\_byte

json\_bin+=(*r*'b"}').encode()

#字节串转为json进行解析

json\_str=json\_bin.decode()

json\_data = json.loads(json\_str,*strict*=False)

print(len(json\_data['a']))  #2

print(json\_data['a']) #b

#恢复出字节串

recv\_bin=json\_data['a'].encode()

print("len(recv\_bin):", len(recv\_bin))#2

print("recv\_bin:", recv\_bin) #b'\x16b'

**自动加载插件设计：**

类最基本的作用是实例化出一个对象，但是有的时候在实例化之前，就需要先和类做一定的交互，这种交互会影响实际实例化的过程，所以必须放在调用构造函数之前。

举例来说，我们的应用有一个插件系统，可以自动从插件目录中加载外部插件并调用。这是一个典型的依赖倒置的场景，典型的代码如下：

*class* Plugin(*object*):

*def* \_\_init\_\_(*self*, *api\_interface*):

        self.\_api = api\_interface

*def* callback(*self*, *event\_type*, *event\_value*):

        """

        Event callback

        """

        pass    # Do nothing

*def* load\_plugins():

    api = create\_api\_interface()

    classes = load\_plugin\_classess()

    for c in classes:

        plugin = c(api)

        register\_plugin(c, plugin)

Plugin是用户自定义插件的基类，它的构造函数接受一个api\_interface的参数，将它保存到实例中。接下来，我们的应用会在合适的event出现时调用plugin的[callback接口](https://www.zhihu.com/search?q=callback%E6%8E%A5%E5%8F%A3&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A537385841%7D)，实现通过插件扩展功能的目标。

随着业务的发展，在应用的第二版，我们重新设计了插件系统：

1. 因为插件数量变得很多，而每个插件可能只关注很少的几种事件，我们希望每个插件在注册的时候就可以声明自己只处理哪些事件
2. 我们设计了全新的API interface，接口和以前的设计思路完全不同。
3. 我们希望兼容旧的插件，使得旧插件不做任何改动就可以继续使用。我们为新旧API设计了Adapter。
4. 基于这次升级的经验，考虑到以后的升级每次都可能使用不兼容的API interface，我们希望插件和应用可以协商API版本

那么基于这样的需求，我们可以发现，我们必须在构造plugin对象之前，就先从类中获取到一定的信息。解决方法之一就是通过classmethod获得类的信息。

设计出如下代码：

*class* Plugin(*object*):

    """

    Legacy plugin (v1)

    """

*def* \_\_init\_\_(*self*, *api\_interface*):

        self.\_api = api\_interface

*def* callback(*self*, *event\_type*, *event\_value*):

        """

        Event callback

        """

        pass    # Do nothing

*class* PluginV2(*object*):

    """

    v2 plugin

    """

*def* \_\_init\_\_(*self*, *api\_version*, *api\_interface*):

        self.\_api = api\_interface

*def* callback(*self*, *event\_type*, *event\_value*):

        """

        Event callback

        """

        pass

    @*classmethod*

*def* capabilities(*cls*, *supported\_versions*):

        if "2.0" in supported\_versions:

            return {"api\_version": "2.0", "register\_events": ["request\_start", "request\_end"]}

        else:

            raise IncompatibleVersionException("API version 2.0 is not supported")

*def* load\_plugins():

    api = create\_api\_interface()

    classes = load\_plugin\_classess()

    for c in classes:

        try:

            if hasattr(c, 'capabilities'):

                cap = c.capabilities(all\_supported\_versions)

            else:

                cap = {"api\_version": "1.0"}

            if cap['api\_version'] == '1.0':

                # 1.0 init

                plugin = c(create\_api\_adapter("1.0", api))

            else:

                # 2.0+ init

                if cap['api\_version'] == CURRENT\_VERSION:

                    plugin\_api = api

                else:

                    plugin\_api = create\_api\_adapter(cap['api\_version'], api)

                plugin = c(cap['api\_version'], plugin\_api)

            if 'register\_events' in cap:

                events = cap['register\_events']

            else:

                events = ALL\_EVENTS

            for e in events:

                register\_plugin\_callback(c, plugin, e)

        except IncompatibleVersionException:

            \_logger.warning("Incompatible plugin detected (%r)", c, *exc\_info*=True)

            raise

可以看到，PluginV2有个capabilites()的classmethod，插件系统在初始化插件之前，首先会调用这个类的方法，通过这个方法的信息判断要如何初始化插件实例。