

# 基础语法

2021年12月30日 12:26

python中最基本的内置数据类型：

整型（整数），浮点型（小数以及科学计数法），布尔型（True和False），字符串型

比较大的整数，比如3000000000，可以写作3000\_0000\_0000，便于看清楚到底有多大

布尔型True是1，False是0，这是他们的本质

比如 b = 3

if b:

print"b" 只要不是空的或者0，都为True

逻辑运算符：优先级not>and>or

Pycharm：多行注释：ctrl+/  
单行、整体缩进（往右）：tab  
整体解缩进（左移）：shift+tab

全局常量、频繁用到的全大写，写在类外边

变量大部分写在类里边

常量、初始化变量等作为类属性

常量全大写，下划线连接

初始化变量首字母大写，下划线连接

一般变量写在类函数里边

下划线方法

类函数里边的变量不能直接访问

类的实例对象，采用全小写+下划线分割的方法，和变量是一样的

函数同样使用下划线原则

类名、文件名采用大驼峰原则

在类中，以\_开头的变量，为私有变量

逻辑运算符也可用在数字之间：

原理是把两个数化成二进制，然后进行运算，然后输出的时候再转化成十进制

需要理解，True和False本质上是1和0

&：如果有0，就都是0

|：如果有1，就都是1

^：异或运算（XOR），如果两个值不一样，就是1，如果相同就是0

python内容相同的字符串是同一个对象，也就是id相同

3.14 = 314\*e-2。这里e相当于一个10的底

a=3

常量即3，a即变量

/浮点数除法，常说的除法

//整数除法

%取余数除法，或者说模运算

\*\*幂

divmod(被除数，除数)函数同时得到商和余数，返回的是一个元组

如果原本是整型的数据，在运算过程中用了除法/等运算，就会将结果视作一个浮点型，就算结果是整数，也会跟个小数点

is 是判断两个标识符是不是引用的同一对象，即比较对象的id

==是判断两者的value是不是相等的

is的效率比==高，因为后者默认调用\_\_eq\_\_方法

python中没有“字符”的概念，就算是一个字符，也表示的一个字符的字符串，而C++不一样，C++中

字符是基本数据类型，字符串是由字符构成的

python中已经被定义的字符串不可变，只能新建一个字符串

input函数，可以打印文本，并将用户的输入当字符串返回

```
s = int(input('请输入数字: '))
```

ord () , chr () 前者把字符变成ASCII编码的数字，后者相反

三个单引号或者双引号可以表示多行字符串

python支持空字符串存在，里面无论啥包括空格都没有，长度为零

双反斜杠\\表示反斜杠本身

续行符，注意“\”后面不能跟任何东西，例如：

```
str1 = 'abc\
de'
```

\t跳格，相当于tab键或4个空格键，常用print里的一个参数：end = "\t"

而当打印东西的时候，如果想结果中的东西转行，就要在其中加上\n

Python字符串中，空格也占一个长度

字符串：

正向搜索:"abc", 这里面a的偏移量（索引值）就是0

反向搜索:"abc", 这里面a的偏移量-3，最右边的开始，为-1

可以用[]来搜索字符串里面的东西

用replace (a, b) 来用b代替字符串中的a

a.replace(a, b), 但实际上只是产生了一个新字符串，a原本还是没有变

区间默认左闭右开，也就是左边第一个是包括进去的，而右边的不包括

切片（slice）操作：

标准格式：[起始偏移量start：终止偏移量end：步长step]，这个步长默认是一

```
a=[11,21,32,44]
```

```
a[:]
```

得到：[11, 21, 32, 44]

```
a[0:4:2]
```

得到：[11, 32]

是每n个（每隔n-1个）取一个字符（也就是跳跃着取的）

[:]提取整个字符串

start和end的值若是负数，则表示倒数，步长为负数，表示从右到左反向提取

repr(a): print(repr(a))可以完整表示出字符串a的组成，包括其中的换行符等

split(), join () 前者用指定的字符（串）分割字符串，后者用指定的字符（串）合并字符串。

在合并字符串时也可以用加号，但是加号效率比join () 来的低下

对于只包含字母、数字和下划线的字符串，若被赋值给两个变量，则相当于把他的地址给了两个不同的变量，所以 a is b就是True。变量赋值本质是数值的地址和变量名建立了联系

len () 序列的长度，可以返回字符串、列表等的长度

a.startswith()判断是不是以输入的字符串开头，返回布尔值，下同

a.endswith()

a.find()找到字符串第一次出现的位置

a.rfind()找到字符串最后一次出现的位置

a.count()计数

a.isalnum()判断所有字符是否全是数字或者字母，返回布尔值

a.strip()去除首尾信息，若无参数则可去除空格、换行符等

a.rstrip()去除末尾信息

a.capitalize()句首字母大写

a.title()所有单词首字母大写

a.upper()全部大写

a.lower()全部小写

a.swapcase()大小写倒换

bin()将数字转化为二进制

hex()将数字转换成十六进制

int()可以将其他进制数字转换成十进制

a.format(... )来格式化字符串

用io模块来修改可变字符串，但实际上字符串仍然不是可变的

```
s = 'hello, lwx'
import io
sio = io.StringIO(s)
sio.getvalue()
'hello, lwx'
sio.seek(7)
7
sio.write('k')
1
sio.getvalue()
'hello, kwx'
```

左右移都是和2有关，左移是乘，左移一位相当于乘2，移两位就乘4，右移则相反

左右移位本质上是变量的二进制形式，左移则在二进制数末尾添0，右移则抹掉最后一位

```
a = 8
a >> 2 (>>为右移符号)
2
a << 1
16
```

常见的序列种类：字符串、列表、元组、字典、集合

在后面加.某函数，即调用某方法。内置函数则不需要这样。

del list[index]删除列表指定位置的元素

基本数据类型（int、float、complex、None、bool、str等）当作实参传入函数时，对形参的修改不会影响到实参；但是当列表、字典、元组等类型当参数传入时，形参会修改实参，效果和地址传递是类似的

list.append(x)将元素x增加到列表尾部，注意没有返回值

list.extend(alist)将列表alist中所有元素并入list尾部

list.insert(index, x)在指定索引的地方插入x

list.remove(x)在列表中删除首次出现的x

list.clear()清空列表中所有元素，变成空列表

list.pop([index])删除并返回list的index处的元素，默认是最后一个元素

list.index(x)返回x第一次出现的位置

list.count(x)计算x在列表中的次数，若返回0则表示x不存在于列表中

len(list)列表元素个数（长度）

list.reverse()反转序列的次序

list.sort()永久排序，所有元素原地（即不改变id）排序，默认升序排序。list.sort(reverse = True)则是降序排序

b = sorted(a) 临时排序，原始数据不变，返回一个原始数据排序的列表

list.copy()返回列表对象的浅拷贝

成员资格判断：x in a，若返回True，则表示x存在于a中

max(), min()分别返回列表当中的最大和最小值，sum（）对列表进行求和。使用这三个函数的列表中只能有数字

列表可以a=[...]直接创建，也可以使用构造函数list()创建，或使用range([start], end, [step])帮助创建列表，其中end为必选

```
a = [x*2 for x in range(5)]
a
[0, 2, 4, 6, 8]
```

列表中套列表，[[20, 90], [77, 88, 67], [9]], 即多维列表，套的越多，维数越高

矩阵可用二维列表来表示

用a[][]...多个索引值来索引元素

```
a = [['lwx', 19, 30000, '北京'],
      ['xy', 19, 25000, '上海'],
      ['lx', 20, 20000, '天津']]
```

```
a[0][0]
'lwx'
```

for循环，可以有两个或多个变量。只要后边序列每次循环的返回值中包含多个变量，如：

```
for name,id in student.item():
    print(name)
    print(id)
```

其中student是一个字典

```
for m in range(3):
    for n in range(4):
        print(a[m][n], end = '\t') #end = "表示不换行继续打印，\t表示四个空格，也就是打印一个后不换行，而是空四格后继续打印
    print() #打印完一行，这东西相当于换行符
```

打印99乘法表

```
for i in range(1,10):
    for n in range(1,i+1):
        print('{}*{}={}'.format(n,i,i*n),end=' ')
    print()
```

第二个print不打印内容，而是起到一个换行的作用

tuple（元组）是不可变序列（列表属于可变序列），只支持切片、连接，索引等操作

用小括号（）（小括号可以省略）或者tuple（）函数来创建元组

如果元组中只有一个元素，必须在后边加上逗号，不然解释器会解释成整型或者字符串等其他东西

无论是tuple（）还是list（）都不能直接接受数字比如list(1,2)便会报错

其他地方元组与列表很相似。另外，实际上元组的访问处理速度比列表快

与整数和字符串一样，元组可以作为字典的键（因为元组是不可变的），列表则不能作为字典的键使用

zip（list1，list2.....）将多个列表对应位置的元素（索引值相同）合并成元组，并返回zip对象，相当于形成一个压缩包了

```
a = [10, 20, 30]
b = [40, 50, 60]
c = [70, 80, 90]
d = zip(a, b, c)
list(d)    #将zip进一步转换成序列便于访问
[(10, 40, 70), (20, 50, 80), (30, 60, 90)]
d
<zip object at 0x000002175B655B00>
```

生成器推导式生成的不是列表也不是元组，而是一个生成器对象，需要进一步转换才能访问

```
s = (x*2 for x in range(5))
s
<generator object <genexpr> at 0x000002175B5E3450> #后面是十六进制的内存地址
tuple(s)
(0, 2, 4, 6, 8)
list(s)
[] #只能访问一次元素，第二次就空了，需要再生成一次
s
<generator object <genexpr> at 0x000002175B5E3450>
```

字典是“键值对”的无序可变序列，字典中的每个元素都是“键值对”

键是不可变且唯一的，而值可以变并且可以重复

字典：a = {'name':'lwx', 'age':18, 'place': '沅江'}

可以通过{}，dict()（在这个里面键不需要加引号）来创建字典

```
my_dict=dict(key1='value1',key2='value2',key3='value3')
```

字符串转字典：可使用eval（前提是该字符串确实可以是字典）

判断字典里是否有某个key，可直接使用'xxx' in dict。曾经的has\_key函数已经被弃用

用a['name']，通过key来查找对象

还可以用zip（）函数来创建

```
k = ['name', 'age', 'job']
v = ['lwx', 18, 'work']
d = dict(zip(k, v))
d
{'name': 'lwx', 'age': 18, 'job': 'work'}
```

事实上，在字典中，若使用：

```
a={'sss':111}
for item in a:
    print(type(item))
```

其中的item就代表一个key

通过dict.fromkeys () 函数创建值为空的字典

```
a = dict.fromkeys(['name', 'age', 'job'])
```

```
a
{'name': None, 'age': None, 'job': None}
```

通过sorted高级函数对字典按键或值进行排序

```
b = sorted(a.items(),key=lambda x:x[1],reverse=False)
print(b)
```

注意：必须取出每个键值对，即item，key代表按哪个来排序，后面的x[1]就代表值

用a.get('key')来访问值

a.items()来列出所有键值对

```
a = {'name':'lwx', 'age':19}
```

如果用像访问每个列表元素那样的方法，比如：

```
for item in a:
```

```
    print(item) 这样只会打印出字典的所有键，而没有值的信息
```

增加键值对

```
a['address'] = '沅江'
```

```
a
{'name': 'lwx', 'age': 19, 'address': '沅江'}
```

序列解包可以用于元组，列表，字典等，可以让我们方便地对多个变量赋值

```
s = {'name':'ys', 'age':'ss', 'super':23}
```

```
a, b, c = s
```

```
a
'name'
```

```
c
'super'
```

```
e, d, f = s.values()
```

```
e
```

```
'ys'
```

```
f
```

```
23
```

h, i, j = s.items() .items()方法是以元组的形式返回键值对

```
h
('name', 'ys')
```

```
j
('super', 23)
```

集合是无序可变，元素不能重复的。实际上集合底层是字典实现，集合中所有元素都是字典的键对象，

因此是不能重复的，创建时如果有重复元素会自动归为一个

注意集合和元组是不一样的

使用{}创建集合

```
a = {12, 32, 32}
```

```
a
{32, 12}
```

```
a.add(99)
```

```
a
{32, 99, 12}
```

```
b = ['a', 'dsf', 'sd']
```

```
c = set(b)
```

```
c
{'sd', 'dsf', 'a'}
```

```
a.remove(32)
```

```
a
{99, 12}
```

```
a.clear()
```

```
a
set()
```

集合的运算

```
a = {1, 23, 'ds'}
```

```
b = {32, 'jj', 9}
```

a|b #并集

```
{'ds', 1, 32, 'jj', 23, 9}
```

a & b #交集

```
set()
```

a - b #差集

```
{'ds', 1, 23}
```

```
a.union(  
a  
set()
```

单分支循环

If 条件表达式:

    语句1/语句块1

双分支循环

If 条件表达式:

    语句1/语句块1

else:

    语句2/语句块2

```
set()  
a - b #差集  
{'ds', 1, 23}  
a.union(b) #并集  
{'ds', 1, 32, 'jj', 23, 9}  
a.intersection(b) #交集  
set()  
a.difference(b) #差集  
{'ds', 1, 23}
```

双分支结构中还可以用三元条件运算符: 条件为真的值 if (条件表达式) else 条件为假的值

```
num=int(input('请输入一个数字: '))
```

```
print(num if num<10 else '数字太大! ') #注意这里else和后边的语句必须有不然后会报错
```

多分支结构: 注意几个分支之间必须要有逻辑关系, 也不能随意颠倒顺序, 因为是从上到下依次执行的

if 条件表达式1:

    语句1/语句块1

elif 条件表达式2:

    语句2。。

elif 表达式3:

    语句3

.

。

。等等

[else :           如果条件都不满足则执行else:

    语句块n]

选择结构的嵌套:

可以在if 语句下继续套娃if, 但是一定要注意缩进, 因为缩进决定了代码的从属关系

```
s=int(input('请输入你的分数: '))
```

```
if s>=60:
```

```
if s>90:
```

```
g='优秀'
```

```
elif 80<s<=90:
```

```
g='良好'
```

```
elif 60<=s<=80:
```

```
g='及格'
```

```
else:
```

```
g='不及格'
```

```
print('成绩是{0}, 等级是{1}'.format(s,g))
```

循环结构: 如果条件符合, 则会反复执行循环体里的语句, 每次执行完都会判断一下条件是否为True, 如果是就继续执行

循环结构中至少应该包含改变条件表达式的语句, 以使循环结束, 不然就会变成一个死循环

while循环和for循环

while 条件表达式:

    循环体语句

for循环

for x in 可迭代对象 (序列、字典、迭代器对象、生成器函数、文件对象)

    循环体语句

---

```
for i in range(1,10):
    s=""
    for x in range(1,i+1):
        s+=str.format('{0}*{1}={2}\t',i,x,i*x)
    print(s)
```

break可以用于while 和 for 循环，用来结束整个循环，如果有嵌套循环则只能跳出最近的一层循环

continue是跳出本次循环一次，相当于不做第n次但是继续做第n+1次并没有跳出整个循环，对于无限次循环来说基本没用

#### while True循环非常实用

else: while 和 for循环可以附带一个[else]语句，如果for、while没有被break中断，则在循环结束时执行else语句，否则不执行。这也就意味着如果无尽循环且没有break，则后面的else无论如何都不会被执行

while 条件表达式:

循环体

else:

语句块

或者

for 变量 in 可迭代对象:

循环体

else:

语句块

循环代码优化:

- 1.尽量减少循环内部的计算量
- 2.嵌套循环中尽量减少内层循环中的计算量
- 3.局部变量查询较快，尽量使用局部变量
- 4.连接字符串尽量使用join () 而不是+

```
str = "-"
```

```
seq = ("a", "b", "c") # 字符串序列 (元组)
```

```
print str.join( seq )
```

```
a-b-c
```

.前面的的是拼接符，后面的是一个一个需要拼接的字符串或者其他东西

这个".join ()"很聪明，如果只输入一个字符串，他会帮你拆开成一个一个字符然后相连。如果想要多个字符串相连，则需要将其先赋予成一个变量，不然join () 中不允许输入三个值

.split()同理

列表进行元素的插入尽量在尾部操作，越往前尤其是开头效率越低，因为列表底层是

```
empNum=0
salarySum=0
salary=[]
while True:
    s=input('请输入员工的薪资（按Q或q结束）： ')
    if s.upper()=='Q':
        print('Complete!')
        break
    if not float(s)>=0:
        print('input wrong!')
        continue
    empNum+=1
    salary.append(float(s))
    salarySum+=float(s)
```

```
print('thenumber of stuffs:{0}'.format(empNum))
print('salary:',salary)
print('averagesalary:',salarySum/empNum)
```

```
import time #测试内循环与外循环之
间的效率差异
```

```
start=time.time()
for i in range(1000):
    result=[]
    for m in range(10000):
        result.append(i*1000+m*100)
```

```
end=time.time()
print('thetimebetweenstartandend:',end-
start)
```

.split()同理

列表进行元素的插入尽量在尾部操作，越往前尤其是开头效率越低，因为列表底层是数组实现，在开头操作会使每个元素都发生变化，使用链表可以解决这一问题

用推导式（生成器）创建序列：

列表推导式

表达式 for item in 可迭代对象 [if 条件判断]

比如

```
a=[x*2 for x in range(20) if x%5 == 0]
```

```
print(a)
```

```
[0, 10, 20, 30]
```

还可以使用双、多循环创造多维列表

```
cells=[(row,col)for row in range(1,10)for col in range(1,10)]
```

字典推导式

{key:value for 表达式 in 可迭代对象 }

类似列表推导，其中也能增加if判断、多个for循环

```
my_text='I love you, my wife, xy and I, forever! '
```

```
print({c:my_text.count(c) for c in my_text})
```

一行代码统计所有字符次数数据

集合推导式（集合使用花括号）

跟字典类似，但只有keys而没有values

元组没有推导式

一个生成器只能运行一次，第一次可以得到数据，但之后就没有数据了

函数是可重用的代码块，函数的作用，不仅可以实现代码的复用，更能实现代码的一致性

一致性是指，只要修改函数的代码，则所有调用该函数的地方都能得到体现

所以一个函数尽量只有一个功能

一个程序由一个个任务组成，函数就是代表一个任务或者一个功能

函数是复用代码的通用机制

函数名实际上就是个变量

函数分类：

内置函数

标准库函数（用import 导入库）

第三方库函数

用户自定义函数

def 函数名([参数列表]):

如果函数中有 return 语句，结束函数运行并返回值

如果没有则返回None

注意：和C++一样，Python的函数在运行过程中，只要遇到了return，则该函数立即停止并返回值，后面的函数代码均不会被执行

def func(a, b): 这里的a, b即为形式参数，其是在定义函数时候时用的，符合命名规则即可在调用函数时传递的参数为实际参数

三个单引号或双引号可以定义多行字符串

可以在函数体第一行使用三个单引号或双引号来定义文档字符串

可以通过print（函数名.\_\_doc\_\_）或者help（函数名），来调用文档字符串

惯例为第一行是简述函数功能，第二行是空行，第三行是函数具体说明

注意文档字符串必须缩进，且只能在函数第一行

```
end=time.time()
print('thetimebetweenstartandend:',end-
start)
```

```
start2=time.time()
for i in range(1000):
    result=[]
    c=i*1000
    for m in range(10000):
        result.append(c+m*100)
```

```
end2=time.time()
print('thetimebetweenstart2andend2:',end2-
start2)
```



尽量多使用文档字符串，文档字符串可以用help（函数名）来打印出来而注释不行  
也就是说，不用看源码就能知道这函数是干什么的

返回值：一是可以返回值，而是可以结束函数的运行，即运行到return即结束函数  
返回值只能有一个，但是这个值可以是列表、字典等东西，以此来包含多个值

当后边有括号时，默认调用这个函数，小括号是函数调用的特征  
定义了test（），令c = test，则c（）与test（）等价  
因为他们指向堆（内存四区之一）中相同的一个东西，id也是一样的

全局变量：在函数和类定义之外声明的变量，作用域为定义的模块  
从定义位置开始直到模块结束，全局变量会降低函数的可读性，应尽量少用

传递参数是不可变对象，例如int、float、字符串等，实际传递的还是对象的引用  
在赋值操作时，由于不可变对象不可修改，系统会创建一个新对象。

浅拷贝copy：不拷贝子对象的内容，只是对拷贝子对象的引用。引用的本质是地址，因此可能修改源对象。这点和C++是一样的  
换句话说，只拷贝最外层的東西  
如果涉及到比如列表中的列表的修改等，就会影响原来的对象，否则是不影响原来对象的  
深拷贝deepcopy：会连子对象的内存全都拷贝一份，对子对象的修改不会影响原对象

传递不可变对象的时候是浅拷贝，注意：不可变对象，比如元组是元组部分不可变，如果元组里面有列表等可修改的，则a[0]这样同样可以修改列表中的东西

#### 参数的类型

位置参数：通过位置来确定参数，函数调用时实参默认按位置顺序传递，需要个数和形参个数相匹配。

默认值参数：我们可以为某些参数设置默认值，这样这些参数在传递的时候就是可选的，默认值参数必须放在其他所有参数后边，当然也可以传递手动设置的参数覆盖默认值。

比如 def a(c, d, e=10)：

命名参数：也称关键字参数，是按照形参的名字传递参数

比如上面定义了a（）之后，  
我print（c=10，d=10），这样直接把关键字和值对应起来  
因此关键字可以无视位置参数

可变参数：可变数量的参数，分两种情况：

- \*一个星号，将多个参数收集到一个元组对象中
  - \*\*两个星号，将多个参数收集到一个字典对象中
- （注意星号是放参数前面）

强制命名参数：在带星号的可变参数后边的新增加的参数，必须是强制命名参数，也就是必须传入关键字参数

因为可变参数将值全给收集完了，导致后面的参数会没有赋值，因此如果不是强制命名参数就会报错

在调用时，直接将参数值标好就行，像关键字参数那样

比如：

```
deff1(*a,b,c):  
    print(a,b,c)  
f1(80,b=9,c=0)
```

#### lambda表达式和匿名函数

该表达式可以用来声明匿名函数，lambda函数是一种简单的、在同一行中定义函数的方法，lambda函数实际上生成了一个新的函数对象

lambda表达式只允许包含一个表达式，不能包含复杂语句，该表达式的计算结果

```
f=lambdab,c:a+b+c  
print(f)  
print(f(2,3,4))
```

```
g=[lambdab:a*2 lambdab:b*2]
```

该表达式可以用米声明匿名函数，lambda函数是一种简单的、在同一行中定义函数的方法，lambda函数实际上生成了一个新的函数对象

lambda表达式只允许包含一个表达式，不能包含复杂语句，该表达式的计算结果就是函数的返回值

基本语法如下（有返回值，给到的变量就类似函数名）：

lambda arg1, arg2.....: 表达式

其中arg N为函数的参数，表达式相当于函数体，lambda函数结果就是就是表达式的运算结果

如：

```
p = lambda a,b:a+b
```

```
p(1,4)
```

结果是5

eval () 函数：功能：将字符串str当成有效的表达式来求值并返回计算值

最常用的是用来去除字符串的引号，将其转变为其他类型

语法：eval (source[, globals[, locals]]) ->value

其中source是一个python表达式或函数compile () 返回的代码对象

globals：可选，必须是dictionary

locals：可选，任意映射对象

用这函数可以动态地改变传递进来的元素，灵活

递归函数：

自己调用自己的函数，类似于数学中的数学归纳法

每个递归函数必须包含两部分：

- 1.终止条件：表示递归什么时候结束，一般用于返回值，不再调用自己
- 2.递归步骤：把第n步的值和与第n-1步相关联

递归函数会创造大量函数对象，过量消耗内存和运算能力，在处理大量数据时，请谨慎使用

递归函数中谨慎使用各种输入，如print，递归次数越多输出越多

```
def coun(a):
```

```
    if a<100:
```

```
        a+=10
```

```
        coun(a)
```

嵌套函数：在函数内定义的函数，只在函数内部调用

其作用为：1.封装：数据隐藏。外部无法访问嵌套函数

2.可避免在函数内部使用重复代码

3.闭包

nonlocal用来声明外层的局部变量，外部的变量不能在里层函数内部轻易修改，除非用nonlocal声明。这样的话外层函数的变量只会在内部被修改，外界的值依然不变。注意只有在函数套娃时用这东西。详见右侧代码

global用来声明全局变量。这样的话，global 变量，在内部的修改也不会影响到全局变量

LEGB规则：python在查找名字时，按照此规则查找：Local（函数或者类方法的内部）>Enclosed（闭包，指的外层嵌套函数）>Global（全局变量）>Built

in（内建，即python中自带的特殊名称）

如果都找不到，就只能报错了

with关键字：用于异常处理，其内部封装了try-except-finally语句，提高了易用性

一般用于公共文件流的处理，如打开文件

经典打开文件语句（不需要手动关闭文件）：with open(xxxxx) as f:

enumerate函数：

将可遍历的一个数据（如列表、元组）组合成索引序列，第二个参数为下标起始值（一般为0）。该函数返回一个enumerate对象，每次用循环取出一个

两元素元组，第一个元素为索引，第二个元素为遍历器中第n个值

如果是嵌套列表，则将最里面的列表作为第二个元素

```
print(f(2,3,4))
```

```
g=[lambda a:a*2,lambda b:b*3,lambda c:c*4]
print(g[0](6),g[1](7),g[2](8))
```

Result:

```
<function <lambda> at 0x00000209694C3E20>
```

```
9
```

```
12 21 32
```

```
eval('print("abc")')
```

比如这里eval () 里面就一字符串

```
def factoria(n):
```

```
    if n==1:
```

```
        return 1
```

```
    else:
```

```
        return n*factoria(n-1)
```

```
for i in range(1,6):
```

```
    print(i, '!=', factoria(i))
```

```
def printName(isChinese,name,familyName):
```

```
    def inner_print(a,b):
```

```
        print('{0}{1}'.format(a,b))
```

```
    if isChinese:
```

```
        inner_print(familyName,name)
```

```
    else:
```

```
        inner_print(name,familyName)
```

```
printName(True,'wx','li')
```

```
printName(False,'Ivanka','Trump')
```

```
def outer():
```

```
    b=10
```

```
def inner():
```

```
    nonlocal b
```

```
    print('inner:b',b)
```

```
    b=20
```

```
inner()
```

```
print('outer:b',b)
```

```
outer()
```

结果： inner:b 10  
outer:b 20

next函数：返回迭代器的下一个项目，主要和生成迭代器iter()函数一起使用

语法：next(iterable[,default])，iterable为可迭代对象，default为可选，即没有下一个元素的时候返回该默认值，如果不设置又没有下一个元素则会报错  
返回值：返回下一个项目

yield：和return类似，都可以返回值，只有带yield的才是真正的迭代器

注意yield必须在函数内部

迭代和递归的区别：递归是重复调用函数本身实现循环，迭代是函数内某段代码实现循环，而迭代与普通循环的区别为：迭代时，循环代码中参加运算的变量同时是保存结果的变量，给下一次迭代做值。循环次数多时迭代效率高。递归中，满足某个条件后逐层返回来结束，迭代是使用计数器结束循环

```
if __name__ == "__main__":
```

代码块

这段代码相当于C++中的主函数，即程序从这里开始运行（如果有的话）

## Python文件操作：

f.open()方法打开文件，返回一个文件对象，用于后续的文件读取操作。注意：打开后文件并没有被加载到内存中。f.close()关闭文件  
(注意readall也可能是读取的一个方法)

f.read()文件中读入整个文件内容，如果是大文件，请谨慎使用

f.readLine()从文件中读入一行内容

f.readlines()从文件中读入所有行，以文件每行的字符作为一个元素，形成一个列表。即可读入文件中全部文本，返回一个列表

f.seek()改变当前文件操作指针的位置，offset的值

f.write()向文件中写入一个字符串或者字节流

f.writelines()将一个元素为字符串的列表整体写入文件，如一个有多字符串的列表，会把所有字符串直接拼在一起写入

如果一个文件正在执行写入操作，那么在没有关闭的情况下，直接用.read()等方法是没有任何效果的

打开模式：

'r'只读模式，如果文件不存在，则返回异常FileNotFoundError，默认值（如果不声明就默认这模式）

'w'覆盖写模式，文件不存在则创建，存在则完全覆盖原文件

'x'创建写模式，如果文件不存在则创建，如果存在就返回FileExistsError

'a'追加写模式，文件不存在则创建，存在则在文件最后追加内容

'b'二进制文件模式

't'文本文件模式，默认值。Python可以以文本和二进制两种方式处理文件，当以二进制打开时，读写按照字节流方式，如果是文本打开，则按照字符串

'+'一般与r/w/x/a一同使用，在原功能的基础上追加同时读写功能

b、t、+可以和上面几种打开方式一起用，如wb就是二进制追加写模式，一般用于图片文件的写入

# 类和对象1

2022年1月5日 1:20

python中一切皆对象，支持继承、封装和多态

任何可以使用 . 调用方法的，都是一个对象

面向对象编程将数据和操作数据相关的方法封装到对象中，组织代码个和数据的方式更接近人的思维，从而大大提高了编程的效率

面向对象是面向过程的升华，最终仍然离不开面向过程，解决问题宏观上用面向对象，将其拆解成一个个小问题后最终还是要面向过程

把对象比作饼干，则类就是生产饼干的模具，当然，类本质也是对象，在python中无处不对象

我们通过类定义数据类型的属性（数据）和方法（行为），即：类将行为和状态打包在了一起，实现了对现实世界的描述，并将某些成员对外开放，某些成员不公开其内部细节。这就是封装

对象是类的具体实例

从一个类创建多个对象时，方法是相同的，但是属性数据是不互通的

类包括属性（变量）和方法（函数），前者是状态，后者是行为

对象包括属性（每个对象维持自己并且和其他对象区分开来的属性）和方法（由同一个类所创立的对象共享），前者同样是状态，后者是行为

类名（参数），即是直接调用类的方法

self表示对象本身，且必须位于第一个参数

构造函数\_\_init\_\_()初始化创建好的对象，初始化的含义即是给实例属性赋值

\_\_new\_\_()方法是用于创建新的对象，但我们一般无需重新定义该对象

一个对象包含id、type、和value，其中value又包括属性和方法

创建对象，我们需要定义构造函数\_\_init\_\_()方法。构造方法用于执行实例对象初始化工作，即对象创建后初始化当前对象的相关属性，无返回值，说白了就是把传进去的值先给赋好

\_\_init\_\_()的要点：

1.名称固定，必须是\_\_init\_\_()

2.第一个参数固定，必须是self（刚刚创建好的实例对象本身）

这一点和C++不一样，C++构造函数名字就是类名，而Python统一是\_\_init\_\_()，且Python的类构造函数第一个参数必须是self，但C++的this不需要写，默认存在

3.构造函数通常用来初始化实例对象的实例属性，

只要是在类里边的函数，第一个参数必须为self

4.通过类名”（参数列表）”来调用构造函数，调用后，将创建好的对象返回给相应变量  
self.name即是对象的属性，而name是形参

可以在外边给对象新加属性，但是这属性只限于这一个对象，通过类新建的对象不会有这些属性

定义实例方法时，跟函数相比多了一个self参数

a = Student()

a.say\_score() = Student.say\_score(a)

上面两者是等价的，一般写左边的那种样式，而解释器翻译的是右边的那种代码

obj表示对象，比如上面的a

dir(obj)可以获得对象的所有属性和方法

obj.\_\_dict\_\_ 对象的属性字典

pass 空语句，比如刚定义一个类，但是还没想好到底要定义啥方法，可以直接填pass，当一个占位

isinstance（对象，类型）判断对象是不是指定类型

当解释器执行class语句时，实际上就会创建一个类对象

类属性是从属于类对象的属性，也称类变量，由于从属于类对象，因此可以被所有实例对象所共享

类属性的定义方式：

类变量名 = 初始值

在类中或者类外边，我们可以通过：“类名.类变量名”来读写

类方法是从属于类对象的方法，类方法通过装饰器@classmethod来定义，格式如下

@classmethod

```
class Student:
    def __init__(self, name, score):
        self.name = name
        self.score = score
```

```
def say_score(self):
    print('{0}的分数是：{1}'.format(self.name, self.score))
```

```
s1 = Student('lwx', 16)
s1.say_score()
```

结果：lwx的分数是：16

```
class Student:
    company = 'xyz' # 类属性
    count = 0 # 类属性
```

```
def __init__(self, name, score):
    self.name = name # 实例属性
    self.score = score
    Student.count = Student.count + 1
```

```
def say_score(self): # 实例方法
    print('my company is:', Student.company)
    print(self.name, 'my score is:', self.score)
```

```
s1 = Student('lwx', 80) # s1是实例对象，自动调用__init__()方法
s1.say_score()
print('一共创建{0}个Student对象'.format(Student.count))
```

```
class Student:
    company = 'super'
```

```
@classmethod
def print_company(cls):
    print(cls.company)
```

```
Student.print_company()
```

```
class Student:
    company = 'super' # 类属性
```

```
@staticmethod
def add(a, b): # 静态方法
    print('{0}+{1}={2}'.format(a, b, a+b))
    return a+b
```

```
Student.add(20, 30)
```

```
def 类方法名 (cls[, 形参列表])
```

```
    函数体
```

要点:

1. @classmethod 必须位于方法上面一行
2. 第一个 cls 必须有, cls (class 缩写) 指的就是类对象本身
3. 调用类方法格式: “类名.类方法名 (参数列表)”。参数列表中不需要也不能给 cls 传值
4. 类方法中访问实例属性和实例方法会导致错误
5. 子类继承父类方法时, 传入 cls 是子类对象而非父类对象

静态方法: python 中允许定义与 “类对象” 无关的方法, 称为 “静态方法”

@staticmethod 静态方法只是名义上归属类管理, 但是不能使用类变量和实例变量, 是类的工具包

放在函数前 (该函数不传入 self 或者 cls), 也就是说这个函数只是名义上在类里面, 其他方面和普通函数没区别。所以不能访问类属性和实例属性

静态方法和在模块中定义普通函数没有区别, 只不过静态方法放到了 “类的名字空间里”, 需要通过 “类调用”。有的时候通过类来调用该方法时更方便一点 (或者说比较容易管理), 所以就是为了图个方便

静态方法通过装饰器 @staticmethod 来定义

```
@staticmethod
```

```
def 静态方法名 ([形参列表]):
```

```
    函数体
```

要点:

1. @staticmethod 必须位于方法上面一行
2. 调用静态方法格式: “类名.静态方法名 (参数列表)”
3. 静态方法中访问实例属性和实例方法会导致错误

如果没有 @staticmethod 声明, 则所有在类里面的函数, 第一个参数会提示是 self

del () 方法称为 “析构方法”, 用于实现对象被销毁时所需的操作, 比如: 释放对象占用的资源, 例如: 打开的文件资源、网络连接等

python 实现自动的垃圾回收, 当对象没有被引用时, 由垃圾回收器调用 del () 方法

我们也可以通过 del 语句删除对象, 从而保证调用 \_\_del\_\_ 方法

系统会自动提供 \_\_del\_\_ 方法, 一般不需要自定义析构方法

当程序结束时默认调用 del, 把你所定义的东西都删了, 只是程序都结束了, 你并不能直白的看见这一过程

\_\_call\_\_ 方法和可调用对象

定义了 \_\_call\_\_ 方法的对象, 称为可调用对象, 即该对象可以和函数一样被调用

python 中方法没有重载

在其他语言中, 可以定义多个重名的方法, 只要保证方法签名唯一即可, 方法签名包含 3 个部分:

方法名、参数数量、参数类型

也就是说, 在 python 中只能以方法名来区分各方法, 所以函数命名一定要唯一

C++ 中函数一般有多个重载, 函数名相同但是参数列表有差异

方法的动态性

python 是动态语言, 我们可以动态地为类添加新的方法, 或者动态地修改类的已有的方法

私有属性和私有方法 (实现封装)

python 对类的成员没有严格的访问控制限制, 这与其他面向对象语言有区别

关于私有属性和私有方法, 有如下要点:

1. 通常我们约定, 两个下划线开头的属性 (方法) 是私有的 (private), 其他为公共的 (public)
2. 类内部可以访问私有属性 (方法)
3. 类外部不能直接访问私有属性 (方法), 比如不能直接用.方法名来直接调用
4. 类外部可以通过 “\_类名\_私有属性名” 访问私有属性, 比如.类名\_方法名来访问

注: 方法本质上也是属性, 只不过可以通过 () 执行而已, 所以此处的私有属性和共有属性, 同时也讲私有方法和公有方法的使用

@property 装饰器 (property 是属性的意思)

这个东西可以将一个方法的调用方式变成 “属性调用”

类实例化的时候, a = 类名() 这个类名的括号必须要

在方法内部，想要访问类的成员变量，必须使用self

self和C++的this指针是十分类似的，都是指向对象本身，只不过C++中this会自己加上去无需手动写，而Python的self是必须自己写在函数内部的，传参的时候可以忽略

如果成员属性没有初始值而只在创建对象的时候才赋值，则可令其=None

这一点C++反而更好，即可以只声明变量而不赋值

格式化字符串，可以直接在{}里填上变量而不需要format语句，只需要在字符串前边加上一个f即可。如：f'我是{self.name}，谢谢'

Python的构造方法：\_\_init\_\_()

在创建类对象的时候会自动执行，且会将传入的参数赋值给成员变量

其中第一个参数是self，后面有几个参数，创建对象的时候就要传入几个参数，用以赋值

Python类内置方法（魔术方法）：

\_\_init\_\_()构造方法

构造函数括号里传进去的参数，是实例化时必须传入的参数

如果要在构造函数中传入类属性，即将类属性赋值给实例属性，则必须使用类名的方式调用

不仅仅是构造方法，如果所有类中的函数，如果想要使用类属性，都必须使用类名来调用，

不管这个类属性跟函数是同一个类还是不同类

\_\_del\_\_()析构函数

\_\_len\_\_()计算长度

\_\_add\_\_()和\_\_sub\_\_()加减法的重构

\_\_new\_\_()创建对象方法，在构造函数之前创建对象

\_\_str\_\_()：字符串方法：通过该方法控制类转换成字符串的行为

\_\_lt\_\_()：小于、大于符号比较

\_\_le\_\_()：小于等于、大于等于符号比较

\_\_eq\_\_()：==符号比较

\_\_call\_\_()：可以将其看作一个普通函数，它的作用是将一个实例化的对象变成可调用的，就像函数那样，此时这个实例的名字就是函数名，\_\_call\_\_()所写的东西都是这个函数的代码

Python的魔术方法，包括但不限于这些。简单地，这些魔术方法就是简化的符号重载，你去手动重载运算符，结果和这个是类似的，只不过这个更简单

```
class Person:
    def __init__(self, ak, bk):
        self.ak = ak
        self.bk = bk
    def __str__(self):
        return f'ak={self.ak}, bk={self.bk}'
a = Person('32', '31')
print(a)
```

可正常输出字符串，如果注释掉\_\_str\_\_，则只会打印对象内存地址

```
class Person:
    def __init__(self, ak, bk):
        self.ak = ak
        self.bk = bk
    def __str__(self):
        return f'ak={self.ak}, bk={self.bk}'
    def __eq__(self, other): # other即代表另一个类
        return self.ak == other.ak & self.bk == other.bk
a = Person(3, 4)
b = Person(1, 2)
print(a == b)
```

```
def __eq__(self, other): # other即代表另一个类
    return self.ak == other.ak and self.bk == other.bk
    # 在Python中&并不是代表和，因此必须用and
a = Person(3, 4)
b = Person(3, 4)
print(a == b)
```

Python中的运算符重载是通过类里的魔法方法来实现的，这一点和C++很不一样。当然，C++里自己进行运算符重载就是为了自定义数据类型



的运算，比如类的运算，Python也可以通过魔法方法来实现这一点，不如说Python更加方便

Python定义私有成员：变量名、函数名前加两下划线\_\_

私有函数和私有变量是无法在类外直接调用的，但是在类内部，可以创建一个新函数作为接口，里面调用类的私有属性，从而实现在外部也可访问类的私有成员

一个类多继承了数个父类，且不想再封装新的方法或者属性了，则在类体里边直接写pass，跳过即可

多继承注意事项：如果有同名的类成员，则默认以继承顺序（从左到右）为优先级，即：先继承的保留，后继承的覆盖

复写：子类继承了父类的成员属性和成员方法后，如果想要修改，那么可以进行复写，即：在子类中重新定义同名的属性或方法即可

复写十分简单，直接给变量赋新值，或者以原函数为名重新定义函数即可

复写之后，默认调用子类复写的成员，如果这时还想要调用父类的成员，就需要特殊方式：

1、通过：父类名.成员 调用

2、使用super()调用：

super().成员

注意子类继承了父类，此时父类的构造方法下的各种属性是无法被继承的，因为父类被继承的时候还没有初始化，因此那些属性也不存在

super函数主要解决继承下的构造函数的问题：使其能同时写自己的构造函数，也能调用父类的构造函数

通过super函数，可以在重写构造函数的同时，调用父类的构造函数。

一般子类需要将父类的一个实例，当作自己构造函数中的参数时会用到

变量注解：

语法： 变量名：数据类型 = 初始值

比如在自定义函数的时候，给参数进行类型注解，则在调用的时候就可以提示应该传什么类型的变量进来，方便自己也方便他人

类型注解是提示性的，就算写错也不会报错

对函数返回值进行注解：在函数的第一行：->数据类型

比如：def func(a: list, b: int) -> int

变量的注解，跟C++那些是反着来的，而且可以不用赋初始值：

a: int

多态

需要复写、变量注释等相关知识

父类的函数直接用pass，类似于C++的虚函数

```
class Animal:
    def speak(self):
        pass
class Cat(Animal):
    def speak(self):
        print('哈哈哈')
```

```
def doSpeak(animal:Animal):  
    animal.speak()  
cat=Cat()  
print(doSpeak(cat))
```

函数指定了需要传入Animal类型，但是实际传入的却是子类，则调用子类的方法

若有虚函数，则父类的唯一作用就是为多个子类实现多态

类属性和实例属性的区别：

类属性是类下面直接赋值的属性，实例属性是在构造函数\_\_init\_\_中初始化的属性

1、类属性相当于全局变量，在内存中只存在一个副本，和C++中类的静态成员变量有些类似。所有类的这个属性都是相同的，而且可以在类外通过类名访问该类属性，从而使所有实例的这个属性全部发生改变。当然也可以通过实例来访问

2、实例属性是每个实例私有的属性，和具体的某个实例对象有关系，一个实例对象和另一个实例对象不共享这些属性

如果实例化以后需要只修改某个实例的某个属性而不影响其他实例的这个属性，则必须使用构造函数：比如tank里面的墙，它的hp就必须是在构造函数中修改，不然一改全改

很明显，类属性消耗的内存资源更少，因此如果所有类共有某个属性，则最好当成类属性而不写在构造函数中



类和对象2

2022年1月10日 23:04

面向对象三大特征

1.封装（隐藏）：

隐藏对象的属性和实现细节，只对外提供必要的方法，相当于将“细节封装起来”，只对外暴露“相关调用方法”

通过“私有属性、私有方法”的方式实现封装，python追求简洁的语法，没有严格的语法级别的访问控制符，更多的是依靠程序员自觉实现

2.继承：

继承可以让子类拥有父类的特性，提高代码的复用性

从设计上是一种增量更新，原有父类设计不变的情况下，可以增加新的功能，或者重写已有方法

父类有的子类都有，但子类可以新加有些东西，而父类没有

python支持多重继承，一个子类可以继承多个父类，继承的语法格式如下：

class 子类类名（父类1，父类2，...）：

类体

如果在类定义中没有指定父类，则默认父类是object类，也就是说object类是所有类的父类，从而定义了有些所有类共有的默认实现，比如 new\_()：创建对象的函数

成员继承：子类继承父类除构造方法之外的所有成员

方法重写：子类可以重新定义父类中的方法，这样就会覆盖掉父类的方法，即“重写”

重写很简单，只需要在子类定义一个名字和父类相同但是内容不同的函数即可

3.多态

多态值而一个方法调用由于对象的不同会产生不同行为

多态是方法的多态，属性没有多态

多态的存在有两个必要条件：继承和方法重写，即子类必须重写父类的方法

查看类的继承层次结构：

通过类的方法mro（）或者类的属性\_\_mro\_\_可以输出这个类的继承层次结构

dir（）查看对象属性

用super（）获得父类的定义（方法），注意是定义不是父类对象

表面上是a+b，本质上是在a.\_\_add\_\_(b)

我们可以重写这些运算符，称为运算符重载

与继承类似，继承是is a

而组合是has a，比如MobilePhone has a CPU.

设计模式是面向对象语言特有的内容，是我们在面临某一问题时的固定做法。

最常用的两个模式：工厂模式和单例模式

工厂模式实现了创建者和调用者的分离，使用专门的工厂类将选择实现类、创建对象进行统一的管理和控制

先创建一个工厂，将所有需要的且相似的类型统一在一起创建，在调用的时候直接从工厂类里边拿就行

单例模式（singleton pattern）的核心作用是确保一个类只有一个实例，并且提供一个访问该实例的全局访问点（类的确可以建立无数个对象，但是有的时候确只需要生成一个对象）

单例模式只生成一个实例对象，减少了对系统资源的开销，当一个对象的产生需要比较多的资源，如读取配置文件、产生其他依赖对象时，可以产生一个单例对象，然后永久驻留在内存中，极大的降低开销

模块：只要是以.py为后缀的文件都可以被称为模块

模块中可以包含：变量、函数、面向对象（类->对象）、可执行代码

使用模块的好处：管理方便、易维护、降低复杂度

直接导入整个模块的方式：

import 模块1，模块2，模块3...

导入后使用：

1.模块名.类

2.模块名.函数名（参数）

3.模块名.变量

二.导入模块中相关的数据

from 模块 import 变量、函数、类等（如果星号的话则表示导入所有）

这种可以直接使用其变量、函数等，而不需要在通过模块名.的方式调用

注意：手动添加全局变量\_\_all\_\_后，这句话将不在是星号导入所有的功能，而导入\_\_all\_\_列表中所包含的功能，注意这些功能必须是以字符串的形式（注意，在python3下不建议使用）

python中的包：包是一个分层次的文件目录结构，它定义了一个由模块及子包，和子包下的子包等组成的

python的应用环境，包中要包含一个\_\_init\_\_.py模块。你创建一个包，他会默认给你创建这个特殊模块

（相当于一个包含了特殊模块的文件类）

包的作用：1.将模块归类，便于整理。2.防止模块名冲突(相同路径下两个模块名不可以重名)

如果在包里，比如lwx这个模块在名为abc的包里，则模块名实际为abc.lwx，这样的话就不会产生冲突使用时不要出了包名

包中的\_\_init\_\_.py是负责初始化的，首次在一个工程中使用包中的模块时，\_\_init\_\_.py模块会被默认执行一次，注意是只有一次

这个模块一般会写一些辅助代码，让你更加方便地使用模块，比如在这个模块中便可以写导入模块的语句，减少代码量

项目的发布：

在新的项目里边，因为模块位于其他文件夹，因此找不到

自定义模块切换项目后不能导入，而系统模块可以继续使用，就是因为路径（path不同）

解决方案：1.将模块所在路径手动加入到sys.path中；

用sys.path.append('项目路径')，其中的项目路径可以直接切到项目所在列表，然后复制路径即可。注意：有的时候会出现路径错误，这是因为路径分隔符有两种表示方式：一种是单斜杠/，另一种是用两个\

ps：这种方法在调用函数的时候不会提示

2.将自定义模块，发布到系统目录中

这种方法，会将自定义模块添加到任何一个新建的项目当中

代码重心出现偏移：去解决异常所需要的代码和时间甚至比解决问题本身还要复杂

这样在编程中是十分不好的，比如说求寿，还要花if、else等语句去做容错，防止被除数

是0什么的，这样其实没必要，这个时候就可以引入异常处理

try:

可能出现的代码

except:

如果出现问题则会执行该代码块

典型代码：

a=input('请输入被除数：')

b=input('请输入被除数：')

if a.isdigit()and b.isdigit():

a=int(a)

b=int(b)

if b!=0:

c=a/b

print('{0}除以{1}的商为{2}'.format(a,b,c))

else:

print('Error:被除数不能为0')

else:

print('Error:数字类型有误')

但是注意！这样一来，虽然能避免抛出错误，但是错误的类型就不太明确了

而且有的时候，会显示：异常子句，也就是except子句过于宽泛

```
class A:
    pass
class B(A):
    pass
class C(B):
    pass
print(C.mro())
```

输出结果包含A，B的层次

#测试has-a的关系，使用组合

```
class MobilePhone:
    def __init__(self,cpu,screen):
        self.cpu=cpu
        self.screen=screen
class CPU:
    def calculate(self):
        print('干你丫的',self)
class Screen:
    def show(self):
        print('显示画面',self)
m=MobilePhone(CPU(),Screen())
m.cpu.calculate()
m.screen.show()
#相当于把两个子类当成了可调用的参数给传进去了
```

```
from random import randint
__all__=['randint']
a=randint(1,9)
print(a)
```

```
class CarFactory:
    '''先要创建一个工厂类'''
    def create_car(brand):
        if brand=='奔驰':
            return Benz()
        elif brand=='宝马':
            return BMW()
        elif brand=='比亚迪':
            return BYD()
        else:
            return '其他品牌'
class Benz:
    pass
class BMW:
    pass
class BYD:
    pass

factory=CarFactory()
c1=factory.create_car('奔驰')
c2=factory.create_car('宝马')
print(c1,c2)
```

定义类时：  
初始化是为自己有参数赋值  
如果没有，那初始化可省略  
而无初始化，则类里的函数不要self.

当范围过于宽泛时，甚至某些拼写错误等也会被掩盖而不会报错，这就离谱了

只有except而没有其他条件，那么就会太过宽泛

当然，这也有解决方法，那就是**except可以有多个异常**：

```
try:
    语句
except 异常1:
except 异常2:
    .....
注意：这里的异常n不能是b = 0这样的东西，必须是像ValueError这样的
如果匹配不上except的错误，就会直接报错，所以，以下是标准代码
最后一个是压箱底的语句（前面异常都不匹配的情况下触发）
a=input('请输入被除数：')
b=input('请输入被除数：')
try:
    a=int(a)
    b=int(b)
    c=a/b
    print(c)
except ValueError:
    print('数据类型有误')
except ZeroDivisionError:
    print('被除数为0')
except Exception:
    print('其他异常')
```

注意：异常必须子类在前，父类在后，比如压箱底的句子就必须放在最后边，不然所有的都会显示其他异常

第二种异常的语法方式：（将多个异常放到except的元组里边，而且要知道元组是没有顺序的）

```
a=input('请输入被除数：')
b=input('请输入被除数：')
try:
    a=int(a)
    b=int(b)
    c=a/b
    print(c)
except(Exception,ValueError,ZeroDivisionError)ase:
    print('遇到异常{0}'.format(type(e)))
```

其他有关错误的代码语法：

```
try:
    pass
except:
    这其中，except是如果出现错误会执行的语句，而else是没有遇到错误会执行的语句，而finally是无论出现错误与否都会执行的语句，像某些关键性的语句，比如文件的关闭等，一般放在这里
    pass
finally:
    pass
子类是可以继承父类的属性的，但是要先调用父类的初始化方法，不然就无法继承
可以通过super函数来调用。
如果父类没有需要继承的属性，也可以不调用父类的初始化方法，而如果要继承父类的属性，就需要使用super函数来调用父类的初始化方法
class MyTank(Tank):
    def __init__(self,left,top):
        super(MyTank,self).__init__(left,top)
```

sys，系统模块，是一个列表，sys.path中存储了一系列的目录，这些目录里存在的模块是可以直接用import导入，既然是列表，那么就可以手动将之前不存在于该目录的文件导入到该路径中

注意：模块是.py文件，但是导入时不需要加.py

```
result=sys.path
result.append('所需要加入的文件的所在位置')
```

导入模块时会默认运行一次

解决方法（以坦克大战为例）：

```
if __name__ == '__main__':
    MainGame.startGame()
```

一个.py文件，可以作为一个单独的文件来执行，也可以导入到其他python文件中

如果不加上上面那行，子进程也会跟着创建子进程，就炸了

所以红字后面跟着的是只有主进程执行的语句

当然如果程序不涉及多进程就不用管这个

is是判断两个变量是不是同一个

==用来判断两个值是否相等，is包括==

复合对象：一个对象的成员变量还是一个（特殊）对象

如list = [[1,2],[2,3]]

进程很多，cpu核数、线程数有限：高并发 -> 见得最多

进程很多，cpu核数、线程也足够：高并行（见得很少，毕竟核数多的cpu少）

多任务：多个进程同时处理

以前的代码是典型的单进程，从上到下依次执行代码

而多进程是一个进程（主进程）中出现一个或多个分支（子进程），分别执行对应的任务

子进程不能再创建子进程，主进程理论上可以创建无数个子进程

实现从单进程到多进程的转变：multiprocessing包，直接import 导入即可使用

但因为这个包里有很多东西，一般使用from multiprocessing import xxx

python写程序的一般方法：

比如要写一个游戏，就要先面向对象分析这个游戏里有些什么东西，有些什么类，将游戏整体分解然后逐步分析每个类有些什么属性，比如位置在哪，大小是多少等等，以及一些函数（方法），比如要实现碰撞或者移动，就要创建相应的方法（函数），随后来调用该方法以实现所需的功能

不知道的方法，比如导入模块里面的方法，去找官方文档，学习即可

导入的不同方法：from xxx import \*

就是把某个包里面的一个个函数都导入，无需使用包的名称即可直接调用包内的函数

而import xxx

就是导入某个包，在使用包内的函数时，需要包名.函数名

类的使用：

- 1、将某些步骤，或者某个实体，抽象成一个类
- 2、类也可以起到管理函数的作用，即将某一类函数都放在这个类里面，可以方便管理，这个时候，类里变~边只有函数，没有构造函数或者属性值，每个函数都是静态函数，不需要加上self参数

# 常用Packages

2022年10月12日 20:26

下载包常用源：

清华源：<https://pypi.tuna.tsinghua.edu.cn/simple>

豆瓣源：<https://pypi.douban.com/simple/>

使用时，语法为：pip install -i 源的URL 要下载的包

例如：pip install -i <https://pypi.tuna.tsinghua.edu.cn/simple> gensim

time库：

time.time() 返回一个浮点数表示的当前时间，人类看不懂

time.ctime(x) 将以浮点数表示的时间x转化为人类看得懂的年月日和小时等

time.sleep(x) 将代码在此暂停x秒

```
print(time.ctime(time.time()))
```

datetime库：

和time模块的功能基本一致

```
from datetime import datetime
```

```
#获取当前时间
```

```
current_time=datetime.now()
```

```
#打印当前时间
```

```
print(current_time)
```

jieba库：

jieba.lcut(s) 对s这一中文字符串进行词语分割，默认精确模式，修改参数cut\_all可更改为全模式，返回列表类型，每个切割成的词语当作列表的一个元素

标点符号会被当作一个词

```
print(jieba.lcut(a))
```

jieba.cut(s) 返回的是一个生成器对象，方便于使用循环取出每一个中文词组；用list()对这个生成器强转，则结果和lcut完全一致

random库：

生成伪随机数，只要随机数种子不变，就可复原产生的随机数

random.seed() 初始化随机种子，其中默认值为当前系统时间

random.random() 生成一个[0.0,1.0)之间（左闭右开）的一个随机小数，有很多位

random.randint(a,b) 生成一个[a,b]之间（左右均含）的一个随机整数

random.sample(pop,k) 从pop中随机选取k个元素，以列表类型返回

random.choices 返回参数序列的随机几个元素，第二个参数是权重（列表形式，和为10），第三个参数k为取样次数

random.choice 返回参数序列的随机一个元素，只有一个参数

request库、urllib3库、scrapy库、lxml：看爬虫板块

pip库：

包名后可以指定版本

pip install 包名 ——安装最新版本

pip install --upgrade 包名 同样可以用来升级pip本身

pip uninstall 包名 卸载

os库：

os库是python标准库，包含几百个函数，提供通用的、基本的操作系统交互功能，常用路径操作、进程管理、环境参数等几类路径操作：os.path子库，处理文件路径及信息

os.mkdir()：创建指定路径，如果路径已存在则报错

os.makedirs(): 可递归地创建多层级目录, 且目录存在时则会跳过  
os.path.isfile(): 判断指定的路径是否为文件  
os.path.isdir(): 判断指定的路径是否为文件夹  
os.path.exists(): 如果目录或文件存在, 则返回true, 不存在返回false  
    用isfile和mkdir来用代码创建文件夹  
os.path.abspath(): 返回文件的绝对路径  
os.path.relpath(): 返回文件相对路径  
os.remove(path): 删除路径下的文件, path不能是目录, 不然将报错  
os.listdir(): 用于返回指定文件夹中包含的文件或文件夹的名称列表  
os.path.join(data\_dir, class\_name): 将data\_dir和class\_name两个路径组合起来, 形成一个完整的路径  
进程管理: 启动系统中其他程序  
环境参数: 获得系统软件硬件信息等环境参数

re库:

是 Python 标准库中的模块, 它提供了对正则表达式的支持, 在处理数据时非常有效  
使用re, 可以:  
    re.split()  
正则表达式: 一种用于匹配和操作文本的工具, 由一系列字符和特殊字符组成

matplotlib: 一个二维数据可视化库

画图时一定要注意数据到底是不是数字类型  
import matplotlib.pyplot as plt  
plt.rcParams['font.sans-serif'] = ['SimHei'] # 指定中文字体防止乱码  
fig, ax = plt.subplots() # 实例化轴对象, 每个轴都是一个实例  
ax.plot(title,value,c='red') # 指定x和y轴数据, 设置曲线颜色  
# 设置标题、x和y轴名称  
ax.set\_ylabel('查看人数(千人)')  
ax.set\_xlabel('项目名称')  
ax.set\_title('可视化',fontsize=30)  
fig.autofmt\_xdate() # x轴标签自动倾斜, 防止标签过长而重叠  
  
plt.show() # 将数据以图表形式展示

csv库: 用来对csv文件的处理

csv文件是有一个首行的、数据以逗号分割的类似表格的文件  
with open('data.csv','w',encoding='UTF-8',newline='') as f:  
    writer = csv.DictWriter(f,fieldnames=list1) # 创建writer, 并指定首行的标签, list1是存折标签名的列表  
    writer.writeheader() # 写入首行  
    writer.writerows(key\_value) # 写入每一行, key\_value是存着小单元的列表, 每个小单元就是csv的一行  
    # 小单元的长度必须和首行长度一致 (毕竟和表格类似)  
  
with open('data.csv') as f: # 由于不是字符串的读写, 因此不能使用'r'或其他读取方式, 只需要打开文件即可  
    reader = csv.reader(f) # 用csv.reader(f)读出内容, 存到reader变量中  
    header\_row = next(reader) # 使用next取出reader的首行  
    data = []  
    for line in reader: # reader每一行就是csv文件的一行 (自动去除了\n等符号)  
        data.append(line) # line中存的数据均为字符串

gzip库: 进行简单的压缩和解压功能

logging: 为Python的标准库, 能够输出日志, 而日志能让我们了解Python的运行情况

logging.basicConfig(format=" ",level=logging.INFO) # 进行设置, 日志将提供基本信息

Python下载第三方库有个很明显的特点：下载某个库时，会下载这个库所有的依赖库，比如下载openai的库，若使用该库，需要使用request库发送http请求，因此下载完openai库之后，还会继续下载request等库

## Numpy包：

numpy中一般都是以数组来作为运算对象的

### 基本使用

```
import numpy as np
array1 = np.array([1,2,3]) #创建初值为[1,2,3]的数组
array2 = np.zeros((3,2)) #创建3*2的全零数组（先行后列）
array3 = np.ones((3,2)) #创建3*2的全1数组
array1.shape() #获取数组的尺寸
array4 = np.arange(3,7) #创建从3到7的数组，左开右闭。升序降序均可
array5 = np.linspace(0,1,5) #返回从0~1等间距的5个数
array6 = np.random.rand(2,4) #生成尺寸为2*4的随机数组，数的范围为0~1
```

### 基本运算

相同尺寸数组，可直接四则运算，相同位置进行加减乘除：

```
a = np.array([2,3,4])
b = np.array([1,2,3])
a+b结果为[3 5 7]
```

### 矩阵乘法运算：

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
result = np.dot(A, B)
print(result)
结果为
[[19 22]
 [43 50]]
```

若a、b为矩阵就是矩阵乘法，若为向量就是计算向量内积  
通常情况下A@B和np.dot(A, B)相同

np.sin(a)、np.cos(a)进行三角函数运算

np.log(a)、np.power(a,2)进行对数和指数运算（2代表平方）

本质仍然是对每个数进行单独运算并输出一个形状相同的数组

a\*5：数组与数做运算，等同于数组中每个数与该数做运算

广播运算：两个不同尺寸数组可以直接运算，但必须满足：其中至少一个数组某个维度为1，或两个数组必须有一个维度相同。广播本质是将元素复制到其它维度上

```
[1,2,3]
广播后
[[1,2,3],
 [1,2,3],
 [1,2,3]]
```

a.min()或a.max()：返回数组中最小或最大的元素

a.argmin()或a.argmax()返回数组中最小或最大元素的索引

a.sum()返回数组所有数据的和

a.mean()或np.median()返回数据的平均值

a.var()、a.std()返回数据的方差或标准差

一个重要的参数：axis

以上这些函数基本都可以添加该参数，axis等于几就代表数组下标第几个数字，如axis=0代表行，即按行操作，以行为单位跨行操

```

作
C = np.array([[1,2,3,4,6],
              [2,4,5,7,9]])
print(C.mean())
print(C.mean(axis=0))
结果
4.3
[1.5 3.  4.  5.5 7.5]

```

获取数组中的元素：和Python列表类似，如a[0,1]表示获取0行1列的数（行列均从0开始）

切片语法：a[0,0:2]，区间左闭右开，若只有一个：则表示获取0行所有列的元素，或者直接写a[0]也可以

## Pandas

基于numpy的数据分析库，提供了大量便于处理数据的函数和方法

数据帧（DataFrame）：pandas的主要数据结构，二维，大小可变，表格型数据，看起来和用excel打开的csv文件类似

数组（Series）：另一种主要结构，一维，由数据和索引组成，类似字典，也可以理解为数据帧中的一列。

利用python代码打开网页文件（默认浏览器打开）：

```

import webbrowser

# 打开本地 HTML 文件
filepath = 'path/to/local.html'
webbrowser.open_new_tab(filepath)

# 打开 URL
url = 'https://www.example.com'
webbrowser.open_new_tab(url)

```

flask：一种轻量级的Python Web框架，方便构建应用程序

在设计完web的基本框架后，可利用flask获取web前端提交的数据，然后将服务处理结果返回前端

# 补充知识

2022年10月12日 21:01

## ctrl+shift+f, 简体繁体切换

Ctrl+c强制暂停正在运行中的代码

json数据格式：是JavaScript Object Notation的缩写，是一种轻量级的数据交换格式，易于程序员阅读和编写，易于计算机解析和生成，其实是JavaScript的子集

特点：

- 1、只能使用双引号
- 2、json数据中，使用{}表示一个json对象，使用[]表示一个json数组，对象类似字典，可包含多个键值对，键值对又可继续嵌套，其中数组可以作为值，即json数组
- 3、json本质为一个字符串，只不过有特定的格式，可以使用特定方法解析

Python使用json语法：先import json

json.dumps(): 将Python对象编码成JSON字符串。这种方法也可以把字典变成json格式

```
a = [2,3,4,5]
jsondata = json.dumps(a)
print(jsondata)
```

json.dump(): 用的较少，但能更方便将数据写入json文件。当然先dumps转化再写入文件也是一样的

```
a = [2,3,4,5]
with open('xyz.json','w') as f:
    json.dump(a,f)
```

json.loads(): 将已编码的json字符串解码成Python对象

```
# jsonData可以是网上得到的json数据
text = json.loads(jsonData)
print(text)
```

json数据格式和Python的对应关系：

JSON	Python
object	dict
array	list
string	unicode
number (int)	int, long
number (real)	float
true	True
false	False
null	None

Unicode编码：原来ASCII只支持127个字符，而Unicode将所有语言统一到一套编码中，这样就不会出现乱码的问题了。现代操作系统和大多数编程语言都直接支持Unicode

所以在网页上直接复制一些东西，可能会出现乱码

python所安装的包都可以在pycharm的python packages里面查看



终端相当于Windows系统的cmd终端，而python控制台则是控制运行、调试、终止代码的地方

Python文件的打包（借助Pyinstaller）

1、这种方法虽然生成的是exe文件，但不是真正的二进制程序，它本质是将py文件、依赖的库以及Python解释器全都打包到一起了，运行的时候，仍然需要启动Python解释器。这样的打包不会对性能优化有一点帮助，反而可能会使速度变慢，好处就是不需要执行的机器安装各种依赖包即可运行

2、安装Pyinstaller：使用pip即可

3、在cmd中执行命令：

pyinstaller 文件名：生成文件，是一个含有exe文件的文件夹

以下是可选的参数

-F：打包 Python 程序为单个可执行文件

-D：打包 Python 程序为一个文件夹

-i：指定exe文件的图标，只适用于 Windows 平台

-n：指定打包后生成文件的名称

-w：禁止命令行弹出

Ctrl+A全选，无论是代码还是office系列的软件都适用

Ctrl+alt+I，格式代码，可以让单行变成多行

可以不实例化类，直接通过类调用类的函数，如MainGame是一个类：

```
MainGame().main()
```

这种方法，注意到类后面跟了括号，即已经实例化了，只是没有赋值给某个变量，和匿名函数思想是类似的，如果懒得实例化，也可以使用这种方法

如果该类有构造函数，则需要在第一个括号里传参

不过最好还是先实例化

python多文件编写

首先要注意的是，python的包名字要求比较严格，不要使用下划线、英文句点等

通过sys.path.append加入其他文件夹的包

编写完成一个.py文件后，直接通过导入包的方式导入，即可通过包名调用里面的类和方法

if \_\_name\_\_ == '\_\_main\_\_': 语句

这段代码的主要作用主要是让该python文件既可以独立运行，也可以当做模块导入到其他文件。当这个模块导入到其他文件的时候，此时\_\_name\_\_的名字其实是导入模块的名字，不是 '\_\_main\_\_'，main代码里面的就不执行了。

简而言之，就是这段代码下的语句，只有在这个文件单独运行时才调用，当作为模块导入其他文件时不调用。但注意，代码并不是从此处开始运行的，而是仍然从上往下

技巧：使用input函数，如果用户直接跳过，则视为“不满足”条件，即用if+input，如果用户输入了就执行下面的，如果没输入就不执行。其他函数有的也能这样用

简而言之，就是返回值或变量为空，均视作不满足条件，如：

```
a = input('指定超时时间：')
```



```
if a:
    socket.setdefaulttimeout(eval(a))
```

三引号：一般用在模块文件开头部分，或是函数第一行。三引号的注释能够在不打开源代码的情况下获取到，更加方便。如果在函数下面使用三引号说明函数用途，别人调用这个函数时，Pycharm会提示函数用途这段话

pycharm如果路径是\，则一定要注意，可能会报错，要么改成\\，要么改成/

## Jupyter Notebook

Jupyter本质为基于网页的用于交互计算的应用程序，类似于iPython以及Python的交互模式（控制台），其可被应用于全过程计算：开发、文档编写、运行代码和展示结果

简而言之，Jupyter Notebook是以网页形式打开，可以在网页页面中直接编写代码和运行代码，代码的运行结果也会直接在代码块下显示的一个程序。如果在编程过程中需要编写说明文档，可以在同一个页面中直接编写，便于作及时的说明和解释

下载：可使用pip install jupyter，更常用的方法为安装Anaconda，之后会自动安装该程序，点击运行后，会自动跳转到Jupyter的网页

注意木星英文名是Jupiter，有点小区别

打开：

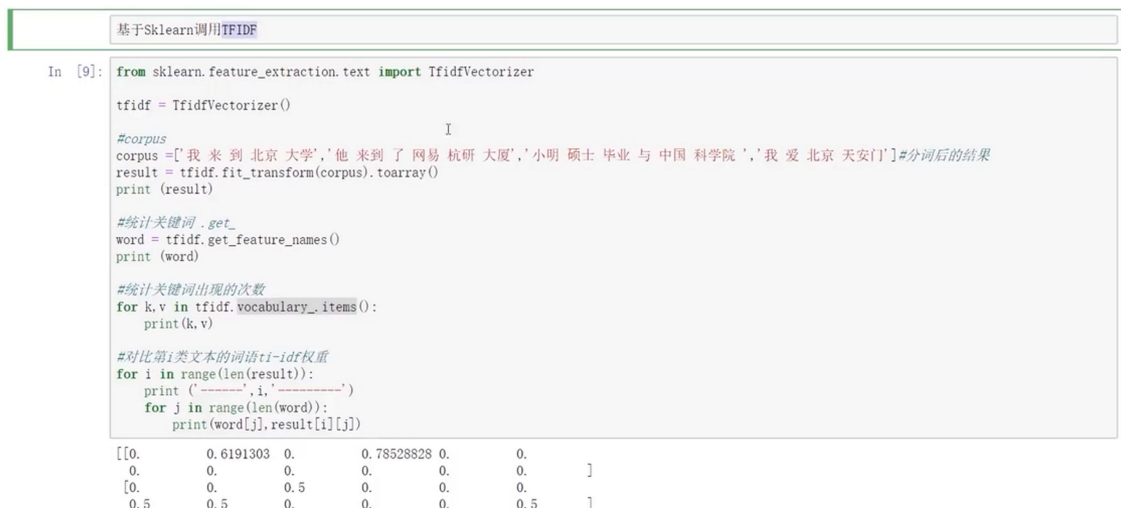
- 1、若通过anaconda安装的，则直接点击jupyter快捷方式即可启动
- 2、pip安装的，在cmd中跳转到python安装路径下的Scripts目录下，输入jupyter notebook，即可在本地启动jupyter

使用：

首先需要加载或打开文件：.ipynb扩展名

每一行都是独立的区域，可选择其为代码区或文本区域等

每一行都可以敲一段代码，然后运行



```
基于Sklearn调用TFIDF

In [9]: from sklearn.feature_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer()

#corpus
corpus = ['我 来 到 北 京 大 学', '他 来 到 了 网 易 杭 研 大 厦', '小 明 硕 士 毕 业 与 中 国 科 学 院', '我 爱 北 京 天 安 门'] #分词后的结果
result = tfidf.fit_transform(corpus).toarray()
print(result)

#统计关键词 .get_
word = tfidf.get_feature_names()
print(word)

#统计关键词出现的次数
for k,v in tfidf.vocabulary_.items():
    print(k,v)

#对比第i类文本的词语ti-idf权重
for i in range(len(result)):
    print('-----',i,'-----')
    for j in range(len(word)):
        print(word[j],result[i][j])

[[0.          0.6191303  0.          0.78528828  0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.5         0.          0.          0.
  0.5         0.5         0.          0.          0.          0.5         1.]
```

如上图所示，说明性文本、代码块、运行结果井然有序，直观且方便

## Pycharm的debug（调试）模式：

在行前面加上红色的断点，然后以debug模式运行，则程序运行到断点的这一行就会暂停下来。

设置好断点，debug运行，然后 F8 单步调试，遇到想进入的函数 F7 进去，想出来在 shift + F8，跳过不想看的地方，直接设置下一个断点，然后 F9 过去。

注意：断点尽量设置在函数、类行

即：一行有断点，则这行会红色显示，debug运行到哪一行，就变成蓝色

F9: 跳到下一个断点

F8: 单步调试, 即从断点开始, 不断往下, 一旦遇到函数或类等就在那调试

tank大战游戏的编写方法, 用的是三层体系结构, 第一层, 创建类, 通过定义各种类, 将其属性、有些什么功能都做好; 第二层, 实现类, 将所有定义的类, 都在MainGame里的专门函数里实现; 第三层, 也就是调用函数, 通过调用实现类的函数, 完成既定编程目标

- 1、超小型编程, 直接敲代码, 可以不定义函数
- 2、小型编程, 可以适当定义函数
- 3、中型编程, 可以创建类, 然后在一个主函数里实现和调用
- 4、大型编程, 可以借鉴tank的三层体系结构
- 5、超大型编程, Python分文件编写, 在其他文件中实现类, 在main中实现和调用。

分析tank得到的结论:

对于单个变量, 可以实现用None来赋值, 类似声明, 而对于多个变量, 比如tank里的多辆敌方坦克, 则使用列表等来存储

当Python导入模块时, 会将模块最顶层(未缩进)的代码全部执行一遍, 比如pygame的源代码里就有一行是print, 导入并运行后自动打印“欢迎使用pygame”

而使用if \_\_name\_\_ == '\_\_main\_\_':语句, 则不会自动执行一遍, 而且仍然可以调用这个语句下面的函数

因此, 无论是写主文件还是分文件(尤其是分文件, 一定要注意), 最好所有的代码都在这个语句下书写, 这不仅仅是一个习惯

```
print('%s is %d years old' % (name, age))
```

```
print('{} is {} years old'.format(name, age))
```

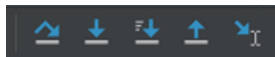
```
print(f'{name} is {age} years old')
```

三种格式化字符串的方法, 第三种是最新的(Python 3.6后支持), 非常方便, 格式化字符串, 如果传入的参数不是字符串类型, 也会自动强制转换成字符串

使用Pycharm进行断点调试:

空行或注释行无法打断点

进入调试模式debug后, 代码会按指定的步骤一步步运行代码, 其中代码前面有红色圆点表示这是一个断点, 调试模式下代码运行到此处会停下。某一行棕红色表示这一行代码是一个断点, 蓝色表示将要运行但还没有运行。在下方的调试控制台处:



分别表示:

步过: 最常用的单步调试, 按照代码运行顺序调试

步入: 跟步入类似, 区别在于: 某一行调用了函数且有断点, 但定义这个函数的里面没有断点, 则运行到这一行, 步过不会进入函数, 而步入就要进去

单步执行我的代码

步出: 从函数中跳出, 继续执行

运行到光标处(用的最多): 注意此光标表示断点的样子, 根据断点来跳跃, 直接跳转到下一个断点  
一般运行到光标处用的最多, 快捷键F9

下面可看到运行到此处时所有的变量的情况

当断点调试发现某一行有异常时，会在这一行停下来，并在行头用一个红色雷电标识。如果此时继续运行断点调试，则会引发报错，这样就知道错在哪了

调试面板左边小工具栏，查看断点，勾选断点集合后点击减号，可全部清除断点

\*args: arguments的缩写，表示位置参数，是一个元组

\*\*kwargs: keyword arguments的缩写，表示关键字参数，必须位于\*args后面，是一个字典

这两者前面带\*是约定俗成的写法

```
def test(first,*args,**kwargs)
test(1,2,3,4,a=3,b=4)
```

函数声明和定义，1是必须填的参数，2、3、4代表\*args接收的参数，a=3，b=4表示\*\*kwargs接收的参数，

一般这种关键字参数虽然没给定关键字，但一般会在函数下第一行的注释里说明，或者举例

args和kwargs组合起来，可以传入任何的参数，这在函数参数未知的情况下是很有帮助的，同时加强了函数的可扩展性

\_\_call\_\_()函数是类里面的一个魔法方法，它将实例化的对象，变成像函数那样的可调用对象，即可以像函数那样去处理这个实例

绝对路径：从根目录下一级级往下的路径

相对路径：相对本文件的路径

写法：注意写入某个路径，是写入这个路径的文件夹下面

./xx——表示xx文件在本文件的同级，爬虫若指定该路径，则把所有东西放在xx下面

./可以不写，单个.代表此文件夹

//xx——表示xx文件在文件夹的里面（下一级）

../xx——表示xx文件在本文件的上一级目录，也就是两个..代表上一级

find只能用于字符串中，而index可以运用在所有有序容器中，而字典和集合没有顺序，因此无法使用find和index函数，其索引值也没有意义

回调函数和函数回调（callback）

定义某个函数，然后在另一个函数中将这个函数名当作参数传进去，并在另一个函数中完成这个函数的调用，该函数即被称为回调函数，该行为称为函数回调。

这样一来，就能通过另一个函数来控制这个函数的调用，请看以下代码：

```
def func1():
    print('pig')
def func2(x):
    x()
func2(x=func1) # 运行结果为：pig
```

在Python中函数名就是一个指针，()是重载的小括号

若是另一个包中的函数，则不能使用包名.函数名来导入

函数只能通过导入整个包或者from语句导入

包中包以及类，则都可以使用

python第三方包的安装

常见的就是使用pip或者conda，使用特定命令安装

也可以先下载包的压缩包，然后解压，在解压后的文件夹路径打开命令行：

- 1、先输入python setup.py build

- 2、再输入python setup.py install

即可成功安装

如果是虚拟环境中的解释器要下这个包，就需要先激活虚拟环境，然后执行和上面相同的操作

exit和quit函数都能直接退出代码，两者作用类似

在cmd中，使用python进入python模式，使用exit()退出python

不要乱删不同版本的python解释器，可能是其它软件的依赖，比如reviews、anaconda等，要删就先找到这个解释器是谁的依赖

要使用python代码往html文件中写入数据，可使用模板引擎来实现：

首先在html文件中定义一个占位符，该变量需要用两层花括号括起来，如：

```
varpoints={{lot}}
```

然后使用python代码将数据填充到占位符中，一个常用的模板引擎是Jinja2

# 常见问题

2022年10月12日 21:04

## DeprecationWarning

一般是所用的库啊啥的版本更新了，然后就会报出这个奇奇怪怪的错误，但实际上不影响运行  
用下面的代码即可防止出现报错

```
import warnings
warnings.filterwarnings('ignore',category=DeprecationWarning)
```

注意！！！不要轻易将函数名以test开头，这样pycharm看到这个函数，会自动调用测试包，就不是直接运行而是在测试了！（其特点是一般的自定义函数面前也会出现一个绿色的小三角，表示可以直接点击运行）

将网络下载的图片插入word文档时报错：

docx.image.exceptions.UnrecognizedImageError

是因为word文档对图片格式要求较为严格，不能直接写入网络图片

通过：

```
import requests
from io import BytesIO
from PIL import Image
response = requests.get(url)
image = Image.open(BytesIO(response.content))
image.save('./temp/temp.jpg')
```

解决

ValueError: invalid literal for int() with base 10报错：

原因：不能直接将带有小数点的数字字符串转化为整数

filenotfounderror：一般是在设置文件名时出了问题，比如文件名中包含如 / 的特殊字符，将其替代即可

keyboardinterrupt：键盘打断了程序执行，一般不用管，是自己终止程序的时候报的错

OSError [WinError5]：安装Python模块的时候可能会出现这个问题，原因是没有权限安装，需要使用管理员权限，不过就算报这个错，一般包都会安装好。如果没有，可以在pip命令最后加上--user，或者以管理员权限打开cmd，跳转到python路径然后安装

未解析的引用+“包名”：这种情况，可能是没有导入相应的包。但还有一种情况，就是代码书写出了问题，比如多了一个小括号等，也会报这个错，因此需要先检查语法

从外部范围隐藏变量：一般是函数里有个变量名，然后外部（包括导入的库、继承的类等）也有有个同名的变量，即一个名字对应两个变量，且一个全局一个局部，所以可能导致解释器识别出错，重命名即可

当多个对象使用append，添加同一个东西时，由于append浅拷贝，大概率会出错  
因此就算是同一个东西，也要用不同的变量名装起来，然后分别append给多个对象

[WinError 5] 拒绝访问：一般在使用代码对某个文件路径进行操作时弹出

原因：对于该路径，权限不足无法操作。或者是函数方法用错了，比如os.remove目标只能为文件但写成了操作某一个目录

解决办法：关闭掉所有和python有关的程序后，找到python解释器，，右键-属性-安全-点击users-把完全控制打勾即可

使用open()时编码报错：

open函数默认使用GBK编码，GBK是含于UTF-8的

当文件全为英文字符，且使用UTF-8编码时，使用GBK编码同样能打开文件，且字符不会出错

若文件中包含英文或其他字符，超出了GBK编码范围时，需要使用UTF-8编码，只需要在open()指定encoding='utf-8'即可

无法加载文件xxxxx，因为在此系统上禁止运行脚本

该问题在pycharm、powershell等均可能出现，原因是电脑策略为restricted

命令行输入：get-executionpolicy，可查看当前政策

以管理员身份运行命令行或powershell，输入set-executionpolicy remotesigned，然后按A或Y，再回车即可

'utf-8' codec can't decode byte：

出现该问题一般是出现了无法进行转换的二进制数据所造成的

一般，只需要将decode的第二个参数，改为'ignore'即可（默认为'strict'）

```
content = response.content.decode('UTF-8','ignore')
```

安装Crypto后，仍然报错：没有Crypto的模块

先将子模块删除，再重装子模块：

```
pip uninstall crypto pycryptodome
```

```
pip install pycryptodome
```

xxx is a required property：某个参数是必填的参数

要么是没填，要么是参数名字填错了，比如少了一个s啥的

# Python环境

2022年11月2日 9:31

环境 (environment, env) : 即Python代码运行环境, 应该包含以下信息:

Python解释器: 解释器在哪: python.exe

Python库的位置: 去哪找并导入安装的第三方库: Lib文件夹, 包括其中的site-packages

可执行程序的位置: pip等命令都是一个小的可执行文件, 它们的位置: Scripts

创建环境, 就是要创建或指定以上信息

环境变量:

操作系统中一个具有特定名字的对象, 包含了一个或多个应用程序将使用的信息。当你要求系统运行一个程序而没有告诉系统完整路径(比如某个文件夹路径中运行python), 系统除了在当前目录下去找此程序外, 还会再path中指定的路径去找, 用户设置环境变量, 来更好运行进程。这也是为什么编程需要配置环境变量, 以及环境变量加多了系统运行会变慢

sys.path

当我们说包的路径就在Lib和site-packages文件夹里时, 虽然大多数情况是这样, 但不准确。

包的搜寻路径是通过Python系统中的一个变量决定的, 也就是sys.path

虚拟环境 (Virtual Environment, venv) : 是Python环境的一个副本

要得到这样的一个副本, 需要: 找个单独的文件夹存起来, 要给他取个名字

这个文件夹的名字就是这个虚拟环境的名字, 在这个文件夹下面有:

一个python.exe

一个Scripts目录

一个Lib目录

注意这个虚拟环境和普通环境有两点不同:

python.exe也放在了Scripts目录下

Lib目录下只有site-packages目录

Python自带venv模块: 用来创建虚拟环境, 也可以用第三程序如anaconda创建

虚拟环境用激活activate, 与去激活deactivate来开关虚拟环境

激活只不过把虚拟环境的Scripts目录临时添加到了PATH环境变量的第一位, 即搜索程序的时候, 先找到了虚拟环境而不是普通环境, 因此用的都是虚拟环境中的东西; 这也解释了为什么把python.exe放在Scripts目录下: 只需要添加一个路径到环境变量中

因此不只有激活才能进入虚拟环境

虚拟环境的作用: 安装不同的包、不同版本的包

由于Pytorch和Pytorch安装在虚拟环境中, 所以解释器和各种第三方包和原来的不通用

若需要安装第三方包, 需要激活虚拟环境, 指定给某个环境安装包:

1、打开anaconda prompt

只有anaconda prompt能够激活虚拟环境, cmd、anaconda powershell prompt、powershell均不可, 无法识别activate命令

2、激活虚拟环境(路径是安装anaconda3的路径): activate 虚拟环境名

我把pytorch安装在的虚拟环境名为: pytorch

3、使用conda install 包名 来安装第三方库, 如果是安装多个包, 则多个包名之间用空格隔开

其他anaconda命令:

conda env list: 打印该路径下所有环境, 带星号\*表示是默认的环境

```
(base) C:\Users\SupernovaGo>conda env list
# conda environments:
#
base            * C:\Users\SupernovaGo\anaconda3
pytorch         C:\Users\SupernovaGo\anaconda3\envs\pytorch
```

conda create -n 虚拟环境名 python=x.x 创建一个虚拟环境

可以不指定python版本, 就相当于只新建一个文件夹并配置一下

**deactivate: 退出该虚拟环境 (不需要环境名)**

conda remove -n 虚拟环境名 --all 删除某虚拟环境, 注意参数顺序

conda config --show-sources 显示当前下载源, 最顶上的就是当前使用的源

conda config --add channels <https://mirrors.aliyun.com/pypi/simple/> 给conda添加国内源, 添加完成后是最高优先级

如果添加了国内源后出现报错, 则需要修改conda的配置文件, 参考: 深度学习-常见问题 中的pytorch的方法

建议对于每个要使用不同库的大工程, 一个工程创建一个环境

如主环境、深度学习环境、使用diffusion环境等, 以牺牲存储空间 (每个环境都要重新下载包) 来换取稳定性



# C++与Python联合编程

2023年7月26日 16:33

若基于Pybind11这个开源项目：

在正式开始之前，对于每个project，需要将pybind11 clone到项目中，作为第三方库进行调用

git clone <https://github.com/pybind/pybind11>

一般新建一个文件夹专门放第三方库，然后将Pybind放进去

完成后，删掉pybind11文件夹里的.git和.github文件夹，那是别人的git

Python调用C++的代码：

原理：将C++代码编译成一个.pyd文件，也就是Python的扩展库，然后导入Python即可

注意：这里由于是调用C++的函数和类，因此含有main()函数的文件就可以不要编译了，直接在Python里写运行逻辑即可

方法：

- 1、编写CMakeLists.txt文件，需要加入pybind11的位置以及需要绑定的源文件.若是单个文件直接写名字即可，若是多个文件可按照下面的格式在{}里写上，以空格隔开

```
# 将pybind11的位置加入，防止找不到这个第三方库
add_subdirectory(externlib/pybind11)
# 第一个参数HelloW是被编译成Python模块的名字，第二是要添加的C++源文件
pybind11_add_module(HelloW ${compiler_list})
```

- 2、在源文件里添加pybind11的头文件

```
#include <pybind11/pybind11.h>
```

如果报错，直接清除错误就行，不影响

这里的pybind11/pybind11.h表示pybind11文件夹下的头文件pybind11.h

- 3、在要build的源文件中绑定函数和类

关于函数的绑定

```
PYBIND11_MODULE(HelloW,m)
```

```
{    // 这里的HelloW是要绑定的函数名称，m是一个指代变量
```

```
    // 第一个是Python里的函数名称，第二个是要绑定的源文件中的函数名称，第三个是函数的描述，可以不写
```

```
    m.def("plusk", &plusk, "A function");
```

```
}
```

关于类的绑定：除了加上pybind11头文件，还要加上py的命名空间

```
#include<iostream>
```

```
#include <pybind11/pybind11.h>
```

```
using namespace std;
```

```
namespace py = pybind11;    //这里py命名空间一定别忘了
```

```
class Plusk
```

```
{
```

```
    int y;
```

```

public:
    Plusk(int input)
    {
        cout << "hell" << endl;
        y = input;
    }
    int add(int x){
        int res = x + y;
        return res;
    }
};          //类定义的结束也别忘了加;
PYBIND11_MODULE(HelloW,m)
{    //在py命名空间下使用Plusk类, "Plusk"是Python里类的名字
    py::class_<Plusk>(m, "Plusk")    // 注意格式
        .def("add", &Plusk::add)    // 类中每个函数都需要这样绑定, 只有类绑定结束了才加;
        .def(py::init<int>());        // 构造函数的绑定, 其中
    <int>是指构造函数返回的是int类型
}    //后面还可以继续加其它类或者函数的绑定

```

4、在cmakelists中指定是构建release类型的项目

```
set(CMAKE_BUILD_TYPE Release)
```

5、检查vscode的状态栏, 使用VS的amd64编译器, 目标是当前的项目, CMake指定release

6、进行build, 得到一个build文件夹, 打开其中的release文件, 有一个项目名称、.pyd结尾的文件, 即Python的拓展库, 将其移动到Python文件位置, 即可直接导入。当然, 也可以移动到Python专门放各种库的地方, 总之是个能找到的路径就可以

C++调用Python代码:

原理: 将Python代码或者脚本嵌入到C++代码中, 并且借助Python的解释器进行解释

这里同样是借助Pybind11, 借助其一个叫embed的头文件

1、首先编写CMakeLists.txt

```

cmake_minimum_required(VERSION 3.26)
project(C++_with_Python)
add_subdirectory(externlib/pybind11)
add_executable(C++_with_Python C++.cpp)
# 下面是用于链接目标 (target) 与库文件或其他目标的命令
# 第一个参数为要链接的目标的名称, 可以是可执行程序、静/动态库等
# 第二个表示链接项仅对当前目标有效, 不会传递给依赖的目标; 第三个表示要链接的库文件或目标, 这里要把embed文件链接
target_link_libraries(C++_with_Python PRIVATE pybind11::embed)

```

2、编写C++源文件

```

#include<pybind11/embed.h>
#include<iostream>
using namespace std;
namespace py = pybind11;
int main(){

```

```

cout<<"C++ with Python"<<endl;
// 初始化Python解释器, 并保持激活状态
py::scoped_interpreter guard{};
// 引入Python的模块的语法
py::module math = py::module::import("math");
// 由于Python中处处皆对象, 所以无论返回什么都是object类型的, math里面
有一个attr (方法, 叫"sqr") , 后面再传参
py::object result = math.attr("sqr")(25);
// 由于Python和C++数据类型不同, 必须经过转换。这里是将object使用cast强
转成float类型
cout<<"answer is "<<result.cast<float>()<<endl;
}

```

3、这之后就可以进行编译了, 选择Unix下的GCC, 先后cmake和make, 得到可执行程序, 再在终端中运行该程序, 若得到结果, 即调用成功

#### 注意事项:

1、在C++调用Python且加载Python的第三方模块时, 注意import语句只能导入模块, 其中不能有类名, 而Python中则可以用from语句, 这一点需额外注意。另外方法只能直接调用而不能链式调用, 例如想使用 `wv.similarity` 的方式, 需要首先获取 `wv` 属性, 然后再调用 `similarity` 方法。如下:

```

py::module models = py::module::import("gensim.models");
py::object w2v = models.attr("Word2Vec");
py::object load_model = w2v.attr("load")("w2v.model");
py::object sim1 = load_model.attr("wv");
py::object sim2 = sim1.attr("similarity")("user", "user");

```

2、编译后的依赖问题:

就像上面的例子, 需要加载`w2v.model`模型, 若用相对路径, 虽然在编译时没问题(编译时是相对源文件的), 但程序运行时就是相对程序本身了, 若源文件和程序不在一个位置则会报错。可以将模型复制一份放到程序所在的位置, 也可以直接用绝对路径

3、可以先写一个Python脚本, 在里面写好所有需要在Python里实现的函数或类, 然后把自己编写的这个脚本当作模块import到C++里去, 这样就可以较为方便地调用里面的函数或类。此时当注意Python导入包的特点: 导入包后包中的代码会自动执行一遍

Python with C++: 不基于Pybind11, 而是使用Python提供的C++头文件与库文件, 将Python作为一个库来调用

首先需要找到Python安装路径下的两个文件夹, 这是Python的API, 用于C的调用

C:\Users\SupernovaGo\AppData\Local\Programs\Python\Python310\libs

C:\Users\SupernovaGo\AppData\Local\Programs\Python\Python310\include

在编译的时候指定这两个目录, 以便编译器能找到库

在CMakeLists.txt中, 可以写:

```
cmake_minimum_required(VERSION 3.26)
```

```

project(py2c)
include_directories("./")
# 指定头文件的路径, 并指定要连接的库文件路径
include_directories("C:/Users/SupernovaGo/AppData/Local/Programs/Python/Python310/include")
link_directories("C:/Users/SupernovaGo/AppData/Local/Programs/Python/Python310/libs")
add_executable(py2c p2c.cpp)
target_link_libraries(py2c PUBLIC python310)    # 如果链接python310出错, 就换成python3

```

而在初始的测试程序中, 可以:

```

#include <Python.h>
#include <iostream>
using namespace std;
int main()
{
    Py_Initialize();    // 初始化python解释器
    PyRun_SimpleString("print('hello')");    // 运行简单的代码
    Py_Finalize();    // 终止python解释器
    return 0;
}

```

注意, 包含Python.h可能会报错, 但只要在cmake中写对了, 是可以编译通过且正常运行的, 可以忽略

无参的调用:

```

#include <Python.h>
#include <iostream>
using namespace std;
int main()
{
    Py_Initialize();    // 初始化python解释器
    if (!Py_IsInitialized())    // 如果初始化失败
    {
        cout << "Python init failed" << endl;
        return 1;
    }
    // 导入sys模块, 用于添加所依赖的路径
    PyRun_SimpleString("import sys");
    // 把模块所在的目录加入到系统路径
    PyRun_SimpleString("sys.path.append('.')");
    // 导入python模块, 不用写后缀名, 返回指向该模块的指针
    PyObject* module1 = PyImport_ImportModule("test1");
    if (module1 == nullptr)
    {
        cout << "module not found" << endl;
        return 1;
    }
    // 获取函数对象, 第一个参数为模块的指针, 第二个为函数名称
    PyObject* func = PyObject_GetAttrString(module1, "say");
    if (!func || !PyCallable_Check(func))
    {

```

```

        cout<< "func not found" <<endl;
        return 1;
    }
    // 调用函数, 这个函数是无参的, 直接用空指针作为参数
    PyObject_CallObject(func, nullptr);
    // 结束调用, 关闭Python解释器
    Py_Finalize();
    return 0;
}

```

如果是有参数的调用, 则需要手动构建参数元组:

```

// 给python函数传递参数, 传递均用元组打包
// 创建Python参数对象, 2代表有2个参数
PyObject* args1 = PyTuple_New(2);
// Py_BuildValue实现参数类型的转换, args1代表传递给的参数元素
// 0表示第一个参数, "i"表示该参数为int类型, 5表示参数的值
PyTuple_SetItem(args1,0,Py_BuildValue("i",5));
PyTuple_SetItem(args1,1,Py_BuildValue("i",4));
// 调用函数, 参数为我们构建的元组
PyObject* res = PyObject_CallObject(func, args1);
int result;
PyArg_Parse(res,"i",&result);    // 将Python的数据转化成C++类型, "i"代表转
化为int

```

```
cout << "返回值是" << result <<endl;
```

注意PyArg\_Parse的最后一个参数必须带&, 也就是引用

如果想要调用Python中的类, 则

```

// 从module1中传进去类Person, 命名为cls
PyObject* cls = PyObject_GetAttrString(module1,"Person");
if (!cls)
{
    cout << "class not found" <<endl;
    return 1;
}
// 构建构造函数的参数, 以实例化这个类, "s"代表string, "f"代表float
PyObject* args1 = PyTuple_New(2);
PyTuple_SetItem(args1, 0, Py_BuildValue("s","jack"));
PyTuple_SetItem(args1, 1, Py_BuildValue("i", 18));
// 实例化这个类
PyObject* obj1 = PyObject_CallObject(cls,args1);
// 从obj中获取函数go
PyObject* func1 = PyObject_GetAttrString(obj1,"go");
if (!func1 || !PyCallable_Check(func1))
{
    cout<<"func not found"<<endl;
    return 1;
}
// 如果这个函数要传参, 则步骤和上面的一致
PyObject_CallObject(func1,nullptr);

```

注意文件之间的依赖问题以及路径

如果cpp源文件中引入了py, py文件毕竟不是一个库, 在链接的时候不会加入到可执行文件中, 因此编译后的exe仍然要依赖于.py文件

注意以下这一行代码:

```
PyRun_SimpleString("sys.path.append('.')");
```

如果添加的是当前的文件夹目录, 则编译得到的exe会在自己所在的文件夹寻找依赖

Cython使用问题:

1、找不到io.h

error: Unable to find vcvarsall.bat

无法运行rc.exe

运行rc出错

无法找到Python.h

```
enum { __pyx_check_sizeof_voidp = 1 / (int)(sizeof(void*) == sizeof(void*)) };
```

# 常用小项目

2023年6月11日 18:58

使用pdf2docx, 将pdf (非扫描件) 转换成word文档

```
import os
from pdf2docx import Converter

def pdf_docx():
    # 获取当前工作目录
    file_path = os.getcwd()

    # 遍历所有文件
    for file in os.listdir(file_path):
        # 获取文件后缀
        suff_name = os.path.splitext(file)[1]

        # 过滤非pdf格式文件
        if suff_name != '.pdf':
            continue
        # 获取文件名称
        file_name = os.path.splitext(file)[0]
        # pdf文件名称
        pdf_name = os.getcwd() + '\\\\' + file
        # 要转换的docx文件名称
        docx_name = os.getcwd() + '\\\\' + file_name + '.docx'
        # 加载pdf文档
        cv = Converter(pdf_name)
        cv.convert(docx_name)
        cv.close()

pdf_docx()
```

使用Tesseract-OCR (需提前安装该软件和附带的chi-sim  
中文包) 等切分图片型pdf并识别文字

```
from PIL import Image
import os
import pytesseract
import cv2 as cv
import fitz

def pdf_image(pdfPath, imgPath, zoom_x, zoom_y, rotation_angle):
    # 打开PDF文件
    pdf = fitz.open(pdfPath)
    # 逐页读取PDF
    for pg in range(0, pdf.page_count):
        page = pdf[pg]
```

```

        # 设置缩放和旋转系数
        trans = fitz.Matrix(zoom_x, zoom_y).prerotate(rotation_angle)
        pm = page.get_pixmap(matrix=trans, alpha=False)
        # 开始写图像
        pm.save(imgPath+'/'+str(pg)+".png")
    pdf.close()

pdf_path = 'C:/Users/SupernovaGo/Desktop/pdf'
img_path = 'C:/Users/SupernovaGo/Desktop/word'
list1 = []

for file in os.listdir(pdf_path):
    # 获取每一个文件的名称
    file_name = os.path.splitext(file)[0]
    os.mkdir(img_path+'/'+file_name)    #创建这个图片的文件夹
    pdf_image(pdf_path+'/'+file_name+'.pdf',img_path+'/'+file_name,5,5,0)    #
    读取每一个pdf文件
    # 依赖opencv
    for img in os.listdir(img_path+'/'+file_name):
        img_name = os.path.splitext(img)[0]
        img = cv.imread(img_path+'/'+file_name+'/'+img_name+'.png')
        text =
        pytesseract.image_to_string(Image.fromarray(img),lang='chi_sim')
        if '工具变量' in text:
            list1.append(file_name)
            break
    print('ok')

print(list1)

```