杜晔的Python笔记

2019.12.22

常见数据结构的成员方法：

> 内容提要：1、字典的方法；2、集合的方法；3、列表的方法

字典的方法

删	dic.clear()、dic.pop(key)、del dic[key]
查	dic.keys()、dic.values()、dic.items()、dic.get(key,333) `解释：如果字典没有key，就返回333`
增	dic.update(dic2) `# 合并2到1上`
拷贝	dic.copy()

集合的方法

删	set.remove(某个元素)
增	set.add(key)、set.update(set2) `# 同字典`
并、交、差	set1|set2、set1&set2、set1-set2

列表的成员方法

删	lis.clear()、lis.pop() `# 默认删除最后一个元素`、lis.pop(2) `# 删除下标为2的元素`、lis.remove('a')
增	lis.append('a')、lis.insert(2,'a')、lis.extend(b) `# b是另一个列表`
查	查下标：lis.index('w')
改	lis[1] = 'w'、lis.reverse() `# 反转`、lis.sort() `# 排序`、random.shuffle(lis) `# 把lis列表的元素打乱`
拷贝	lis.copy()
统计	lis.count('w') `# 统计‘w’元素的个数`

列表操作相关：

> 内容提要：
>
> 1.1、input字符串存入列表；1.2输入存进二维列表；2、把列表的元素用空格做间隔输出；3、倒叙生成列表及步长；4、打印下标；5、统计各个元素出现的次数；6、生成二维列表；7、用sum给列表降维；8、zip合并多个列表；8.2、列表生成式合并2个列表；9、逆时针、顺时针旋转矩阵；10、enumerate() 给元素加下标；11、reduce函数/filter函数；12、排列组合；13、切片操作

删	lis.clear()、lis.pop() `# 默认删除最后一个元素`、lis.pop(2) `# 删除下标为2的元素`、lis.remove('a')
增	lis.append('a')、lis.insert(2,'a')、lis.extend(b) `# b是另一个列表`
查	查下标：lis.index('w')
改	lis[1] = 'w'、lis.reverse() `# 反转`、lis.sort() `# 排序`、random.shuffle(lis) `# 把lis列表的元素打乱`
拷贝	lis.copy()
统计	lis.count('w') `# 统计‘w’元素的个数`

input输入的是字符串，转化成列表：

```
# 功能：input输入的是字符串，转化成列表。
# split()   默认空格,换行\n,制表符\t分割
#map(square, [1,2,3,4,5])   # 计算列表各个元素的平方  square是函数名
line = input()
l = list(map(int, line.split()))
print(l,type(l))
```

```
2 3 4 5
[2, 3, 4, 5] <class 'list'>
```

输入存进二维列表：

```
arr = []
for i in range(4):        # 使用for循环操作
    arr.append(list(map(int,input().split())))
print(arr)
```

```
1 2 3 4
2 3 4 2
2 1 3 4
1 5 4 6
[[1, 2, 3, 4], [2, 3, 4, 2], [2, 1, 3, 4], [1, 5, 4, 6]]
```

把列表的元素用空格做间隔输出：

```
a = [1,2,3]
c = [str(i) for i in a]
print(c)                  # 字符形式组成的列表
print(' '.join(c))        # 用空格连接
```

```
['1', '2', '3']
1 2 3
```

倒叙生成列表：（使用-1步长）

```
a = list(range(5))       # 生成5个数（自然顺序）
print(a)
b = list(range(1,5,2))   # 从1到4（因为前闭后开），步长为2
print(b)
c = list(range(5,1,-1)) # 从5到2（因为前闭后开），步长为-1，即每次+负1
print(c)
```

```
[0, 1, 2, 3, 4]
[1, 3]
[5, 4, 3, 2]
```

打印列表中某个元素的下标：

```
a = [2,3,'f',99,'tttr']
print(a.index('f'))       # 打印元素'f'的下标
```

```
2
```

统计列表中不同元素的个数:Counter函数

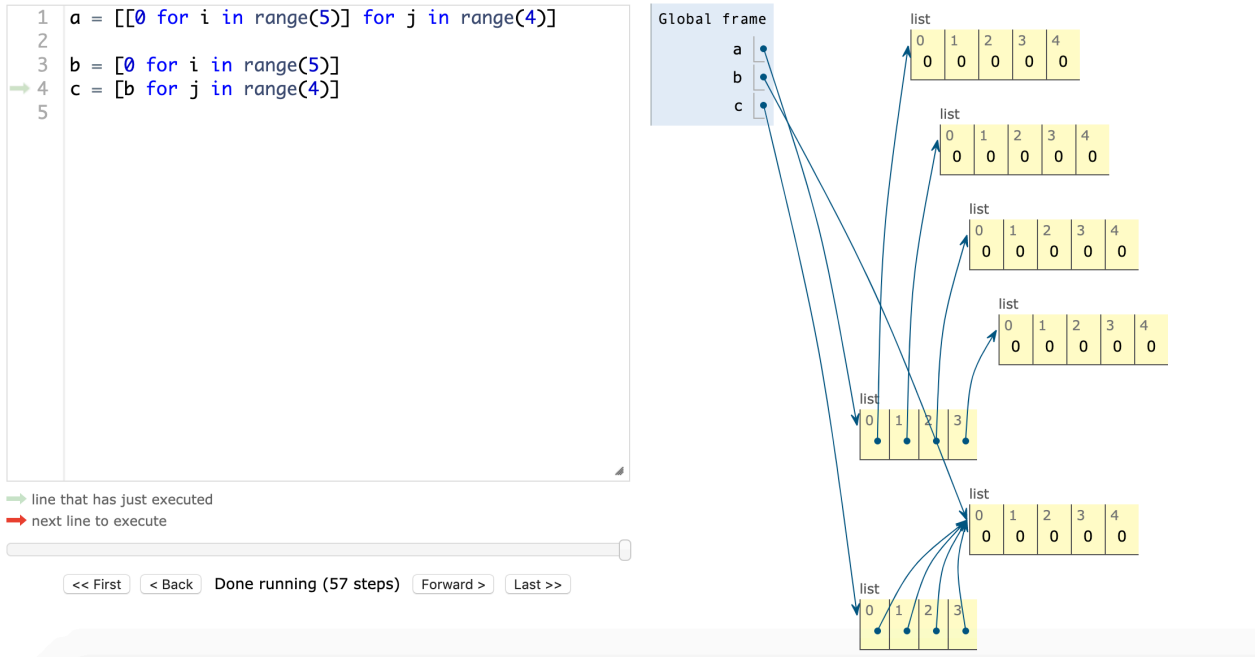
```
#知识点Counter函数

import collections
numbers = [2,2,2,3,4,6,6,7,5]
c = collections.Counter(numbers)
print(c,type(c))
c = dict(c)                                    # 转换成字典格式
print(c,type(c))
c2 = sorted(c.items(),key = lambda m:m[0])     # 以key排序
print(c2)
```

```
Counter({2: 3, 6: 2, 3: 1, 4: 1, 7: 1, 5: 1}) <class 'collections.Counter'>
{2: 3, 3: 1, 4: 1, 6: 2, 7: 1, 5: 1} <class 'dict'>
[(2, 3), (3, 1), (4, 1), (5, 1), (6, 2), (7, 1)]
```

！！生成二维列表：

注意图中两种生成方式的区别！



巧用sum函数给列表降维：

```
# newlist = sum(oldlist,[])
# 效果是 oldlist 中的子列表逐一与第二个参数相加，而列表的加法相当于 extend 操作，
# 最终结果是由 [] 扩充成的列表。即，空列表变成了含有oldlist中所有子元素的一维列表。

# 这里有两个关键点：sum() 函数允许带两个参数，且第二个参数才是起点。
v = [[True, True, True, True],
     [False, False, False, False],
     [False, False, False, False],
     [False, False, False, False],
     [False, False, False, False]]
print(sum(v,[]))
print(sum(sum(v,[])))              # 打印有多少个True
```

```
[True, True, True, True, False, False, False, False, False, False, False,
False, False, False, False, False, False, False, False, False]
4
```

合并多个列表为元组列表:zip函数

```
# 合并后，元素个数和最短的列表一致。
cc = [200,600,100,180,300,450]
vv = [6,10,3,4,5,8]
bb = [33,555,3,1,4,5]
nn = zip(cc,vv,bb)             # 使用zip函数
for i in nn:
    print(i)
```

```
(200, 6, 33)
(600, 10, 555)
(100, 3, 3)
(180, 4, 1)
(300, 5, 4)
(450, 8, 5)
```

列表生成式：组合两个列表

```
color = ['black','red','white']
size = ['s','m','l']
tshirt = [(i,j) for i in color for j in size]
print(tshirt)
```

```
[('black', 's'), ('black', 'm'), ('black', 'l'), ('red', 's'), ('red', 'm'),
('red', 'l'), ('white', 's'), ('white', 'm'), ('white', 'l')]
```

zip：

```
a = [(1, 4), (2, 5), (3, 6)]
list(zip(*a)) # 与 zip 相反，*a 可理解为解压，返回二维矩阵式
```

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

逆时针旋转矩阵：

```
matrix = list(zip(*matrix))[::-1]   # 90度逆时针旋转剩下的矩阵元素
```

顺时针旋转矩阵：

```
matrix = list(zip(*matrix[::-1]))   # 90度顺时针旋转剩下的矩阵元素
```

enumerate() 函数:给元素加下标

> 用于将一个可遍历的数据对象组合为一个索引序列，同时列出数据和数据下标

```
a = ['Spring', 'Summer', 'Fall', 'Winter']
print(list(enumerate(a)))
print(list(enumerate(a,start=1)))  # 下标从1开始
```

```
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

reduce()函数：

> 用传给 reduce 中的函数 function先对集合中的第 1、2 个元素进行操作，得到的结果再与第三个数据用 function 函数运算，最后得到一个结果。

```
from functools import reduce
a = [1,2,3,4]
b = reduce(lambda x,y:x+y,a)
print(b)
```

```
10
```

filter()函数：

> 过滤序列，过滤掉不符合条件的元素，返回由符合条件元素组成的新列表

```
a = list(filter(lambda x:x%2 == 0,range(1,10)))   # 保留偶数
print(a)
```

```
[2, 4, 6, 8]
```

题目：一行代码实现对列表a中的偶数位置的元素进行加3后求和?

```
a = [1,2,3,4,5]
sum(map(lambda y:y+3,filter(lambda x:a.index(x)%2==0,a)))
```

```
18
```

题目：将列表a的元素顺序打乱，再对a进行排序得到列表b，然后把a和b按元素顺序构造一个字典d。

```
import random
a = [1,2,3,4,5]
b = a[:]
random.shuffle(b)
print(a,b)
dict(zip(a,b))
```

```
[1, 2, 3, 4, 5] [3, 1, 5, 4, 2]
{1: 3, 2: 1, 3: 5, 4: 4, 5: 2}
```

全排列&组合：permutations,combinations

```
from itertools import permutations,combinations
c = [1,2,3]
lisss = list(permutations(c,2))    # 排列
lisss2 = list(combinations(c,2))   # 组合
print(lisss)
print(lisss2)
```

```
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
[(1, 2), (1, 3), (2, 3)]
```

operator 里的 itemgetter：

> 对字典组成的列表排序，按照字典的某一个/几个字段。

```
rows = [
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1003},
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
    {'fname': 'Big', 'lname': 'Aones', 'uid': 1004}]
from operator import itemgetter
rows_sortby_fname = sorted(rows,key=itemgetter('fname'))  # 按一个字段排序
print(rows_sortby_fname)
print('--')
rows_sortby_fname2 = sorted(rows,key=itemgetter('fname','lname')) # 按2个字段排序
print(rows_sortby_fname2)
```

```
[{'fname': 'Big', 'lname': 'Jones', 'uid': 1003},
 {'fname': 'Big', 'lname': 'Aones', 'uid': 1004},
 {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
 {'fname': 'John', 'lname': 'Cleese', 'uid': 1001}]
 --
[{'fname': 'Big', 'lname': 'Aones', 'uid': 1004},
{'fname': 'Big', 'lname': 'Jones', 'uid': 1003},
{'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}]
```

感受slice切片操作：

```
a = slice(5,8,2)
b = range(10)
print(list(b[a]))    # 方括号
print(a)             # 看看a长什么样子
```

```
[5, 7]
slice(5, 8, 2)
```

字典的操作相关：

> 内容提要：
>
> 1.1、字典按value/key排序；1.2、dic.values()；1.3、遍历字典dic.keys()；2、collections.defaultdict()默认好了字典的值的数据类型；3、update合并字典；4、zip实现字典的键值反转；5、zip合并不同类型，只要可迭代就行

删	dic.clear()、dic.pop(key)、del dic[key]
查	dic.keys()、dic.values()、dic.items()、dic.get(key,333) `解释：如果字典没有key，就返回333`
增	dic.update(dic2) `# 合并2到1上`
拷贝	dic.copy()

对字典按value、key值排序：

！！sorted之后字典变成了元组构成的列表

```
#知识点：sorted函数。sorted()的参数，链接很详细。
# ！！sorted之后字典变成了元组组成的列表。
#https://www.runoob.com/python/python-func-sorted.html
dicc = {9149: 0, 9150: 26, 9151: 24, 9158: 24, 9153: 25}
dicc2 = sorted(dicc.items(),key=lambda m:m[1])  # 按value排序
dicc3 = sorted(dicc.items(),key=lambda m:m[0])  # 按key排序
print(dicc2,type(dicc2))
print(dicc3,type(dicc3))
```

```
[(9149, 0), (9151, 24), (9158, 24), (9153, 25), (9150, 26)] <class 'list'>
[(9149, 0), (9150, 26), (9151, 24), (9153, 25), (9158, 24)] <class 'list'>
```

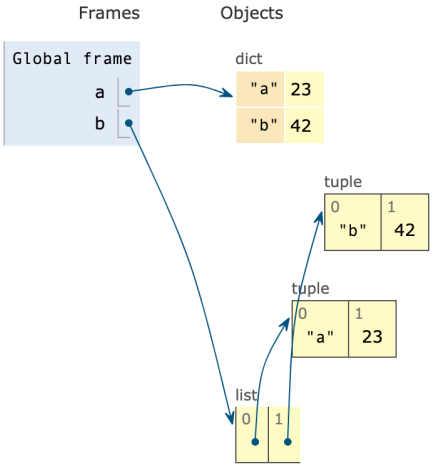
再看个例子：

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  a = {'a':23,'b':42}
2  b = sorted(a.items(),key=lambda m:m[1])
3  print(b,type(b))
4  print(b[0],type(b[0]))
```

Print output (drag lower right corner to resize)

```
[('a', 23), ('b', 42)] <class 'list'>
('a', 23) <class 'tuple'>
```



输出字典的值到 【列表】： dic.values()

```
dic = {2:[3,7],4:[6,5]}
dic2 = {4:6,2:3}
print(dic.values())
print(dic2.values())
```

```
dict_values([[3, 7], [6, 5]])
dict_values([6, 3])
```

遍历字典的误区：

```
# 在使用上，for key in a和 for key in a.keys():完全等价
# 而且在3.几版本之后，字典是有序的
a = {0:'a',33:'b',2:'c'}
for i in a:
    print(i)
```

```
0
33
2
```

collections.defaultdict():

> 生成一个字典，并默认好了字典的值的数据类型，如collections.defaultdict(list) 添加新的键，其值都是默认放在列表里。

```
from collections import defaultdict
dic = defaultdict(list)     # 使用
dic[2].append(3)
dic[2].append(4)
dic[2].append(3)
print(dic)
dicc = dict(dic)
print(dicc)
```

```
defaultdict(<class 'list'>, {2: [3, 4, 3]})
{2: [3, 4, 3]}
```

合并两个字典：update方法

```
d1 = {'dog':3,'cat':5}
d2 = {'miemie':2,'gigi':7}
d1.update(d2)               # 合并
print(d1)
```

```
{'dog': 3, 'cat': 5, 'miemie': 2, 'gigi': 7}
```

使用zip实现字典的键值反转：

```
d1 = {'dog': 3, 'cat': 5, 'miemie': 2, 'gigi': 7}
d2 = dict(zip(d1.values(),d1.keys()))
print(d2)
```

```
{3: 'dog', 5: 'cat', 2: 'miemie', 7: 'gigi'}
```

zip合并不同类型，只要可迭代就行：

```
a = [1,2,3]
b = 'edw'
c = list(zip(a,b))
d = dict(zip(a,b))
print(c)
print(d)
```

```
[(1, 'e'), (2, 'd'), (3, 'w')]
{1: 'e', 2: 'd', 3: 'w'}
```

集合的操作相关：

> 内容提要：
>
> 1、生成空集合；2、集合添加元素；

删	set.remove(某个元素)
增	set.add(key)、set.update(set2) `# 同字典`
并、交、差	set1|set2、set1&set2、set1-set2

生成空集合：

```
a = set()  # 不能用{}, {}是用来生成空字典的
```

集合的添加元素：

```
a.add('ws')
```

元组的操作相关：

> 内容提要：
>
> 1、元组的特点描述；2、元组拆包；3、元组作为可变参数；4、具名元组；5、字典推导式

元组的特点描述：

> 1.不可变列表 2.没有字段名的记录（关注他的数量和位置信息）

```
name,age,city = ('wang',12,'nanjing')
print(name,city,age)
```

```
wang nanjing 12
```

元组拆包：

```
tem = ('wang',12,'nanjing')
name2,age2,city2 = tem   # 神奇吧，元组拆包
print(name2,age2,city2)
```

```
wang nanjing 12
```

用*运算符把可迭代对象（如元组）拆开作为函数的参数：

```
def func(x,y):
    return (x+y)

t = (2,5)
print(func(*t))    # 就是以前见到的*args，不知道有几个参数的情况
```

```
7
```

具名元组：nametuple

```
from collections import namedtuple
city = namedtuple('City','name country population')  # 头
beijing = city('beijing','china','1000')             # 实例化
print(beijing.name)                                  # 使用字段名
print(beijing[1])                                    # 使用位置
```

```
beijing
china
```

字典推导式：可迭代对象是元组

```
a = [('w',2),('do',3),('ss',4)]
b = {i:j for i,j in a}
print(b)
```

```
{'w': 2, 'do': 3, 'ss': 4}
```

字符串的操作相关：

> 内容提要：
>
> 1、修改字符串；2.1、.index()方法；2.2、.find()方法；3、字符串的sort和sorted；4、反转字符串；5、统计字符串元素；6、endswith方法；7、字符串大小写的转化

修改字符串：

字符串是不可变量，无法直接修改：

```
# 思路2：使用replace()函数。repalce()还有第三个参数，表示替换几次。
a = 'we are happy.'
a= a.replace(' ','%20')
print(a)
```

```
we%20are%20happy.
```

```
#思路1：先把字符串转成列表，修改后，再拼接回去

# 知识点：join ()
#下边函数的功能是将字符串中的空格改成'%20'
a = 'we are happy.'
b = list(a)
print(b)
for i in range(len(b)) :
    if b[i] == ' ':
        b[i] = '%20'
print(b)
a = '--'.join(b)
print(a)
```

```
['w', 'e', ' ', 'a', 'r', 'e', ' ', 'h', 'a', 'p', 'p', 'y', '.']
['w', 'e', '%20', 'a', 'r', 'e', '%20', 'h', 'a', 'p', 'p', 'y', '.']
w--e--%20--a--r--e--%20--h--a--p--p--y--.
```

.index()方法：

```
a = 'dogdog'
print(a.index('g'))
print(a.index('og'))
print(list(map(a.index,a)))
```

```
2
1
[0, 1, 2, 0, 1, 2]
```

.find():

与.index（）的区别：

找不到的时候，find（）返回-1；index（）找不到就报错。

字符串的sort和sorted:

```
# 字符串没有sort()方法
s = 'wre'
s.sort()
```

```
AttributeError: 'str' object has no attribute 'sort'
```

```
# ！但是，可以使用sorted对str进行排序，
# 生成list格式
s = 'wre'
print(sorted(s))
```

```
['e', 'r', 'w']
```

反转字符串：

```
a = 'dogg'
print(a[::-1])                # 方式1
print(''.join(reversed(a)))   # 方式2
```

```
ggod
ggod
```

统计字符串中有多少个某字母/字符串：

```
a = 'wwwderft'
print(a.count('w'))
print(a.count('er'))
```

```
3
1
```

题目：

> 统计字符串A中有多少个字符也在字符串B中可以找到
>
> 分析：
>
> python自带的string.count(char)函数的作用是统计一个字符串string含有字符char的数量。在本例中strB相当于char的一个参数列表["a", "A"], map函数先统计strA中字符a的数量，再统计strA中字符A的数量，获得列表[1, 3], 然后将它们相加，即可获得字符串A中总共有多少字符可以在B中找到。

```
strA = "aAAAbBCC"
strB = "aA"
print(sum(map(strA.count,strB)))
```

```
4
```

判断字符串是否以‘rtr’结尾：

```
a = 'wdsdfertr'
print(a.endswith('rtr'))
print(a.endswith('twr'))
```

```
True
False
```

关于字符串大小写，小小的总结：

```
a = 'i love you. Do you know? BIG'
# 每个单词的首字母大写
title = a.title()
print('1、',title)
# 所有字母大写
upper = a.upper()
print('2、',upper)
# 所有字母小写
lower = a.lower()
print('3、',lower)
# 字符串的第一个字符大写，注意不是第一个字母
capitalize = a.capitalize()
print('4、',capitalize)
# 所有字母大小写互换
swapcase = a.swapcase()
print('5、',swapcase)
```

```
1、 I Love You. Do You Know? Big
2、 I LOVE YOU. DO YOU KNOW? BIG
3、 i love you. do you know? big
4、 I love you. do you know? big
5、 I LOVE YOU. dO YOU KNOW? big
```

Python 面向对象：

> 内容提要：
>
> 1、常见的类；2、类变量和实例变量对比；3、含有类变量；4、含有类方法；5、私有属性；6、私有属性的访问；7、普通继承；8、super继承；9、类中重构**repr**和**str**；10、*__new__*和*__init__*；11、__call__方法；

常见的类：

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

    def show(self):
        info = '姓名:{},成绩:{}'.format(self.name,self.score)
        return info

a = Student('de',33)
print(a.name)
print(a.score)
print(a.show())
```

```
de
33
姓名:de,成绩:33
```

类变量和实例变量：

这个程序简单明了：

```
class Student():
    age = 0
    name = 'stu'
    # age,name是类变量
    def __init__(self,age,name):
        self.age = age
        self.name = name
    # 访问实例变量(用self.age  self.name)

student1 = Student(18,'hello')
print(student1.name)
# 打印实例变量，输出hello
```

```
print(Student.name)
# 打印类变量，输出stu
```

含类变量：（所有实例都可访问，即公共的）

```
class Student():
    number = 0                  # 类变量
    def __init__(self,name,score):
        self.name = name
        self.score = score
        Student.number += 1    # 修改类变量：每有一次实例化，+1

    def show(self):
        print('姓名:{},成绩:{}'.format(self.name,self.score))

a = Student('de',33)
print(Student.number)           # 可通过Student访问
print(a.number)                 # 也可通过实例访问，但是实例只能访问，不能修改
```

含类方法：（所有实例都可访问，即公共的）借助@classmethod

```
class Student():
    number = 0
    def __init__(self,name,score):
        self.name = name
        self.score = score
        Student.number += 1

    def show(self):
        info = '姓名:{},成绩:{}'.format(self.name,self.score)
        return info

    @classmethod
    def total(cls):
        info = '创建了{}个实例。'.format(cls.number)
        return info

a = Student('de',33)
print(Student.total())          # 可通过Student访问
print(a.total())                # 也可通过实例访问
```

```
创建了1个实例。
创建了1个实例。
```

私有属性：前边加上双下划线__。不能通过实例直接访问

```
class Student():
    def __init__(self,name,score):
        self.name = name
        self.__score = score    # 私有属性

    def show(self):
        info = '姓名:{},成绩:{}'.format(self.name,self.__score)
        return info

a = Student('de',33)
print(a.name)                   # OK
print(a.show())                 # OK
print(a.__score)                # 报错
```

```
de
姓名:de,成绩:33
AttributeError: 'Student' object has no attribute '__score'
```

又想通过a.score访问私有属性：借助@property

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

    def show(self):
        info = '姓名:{},成绩:{}'.format(self.name,self.__score)
        return info
# 新增
    @property
    def score(self):
        info = self.__score
        return info

a = Student('de',33)
print(a.name)
print(a.show())
print(a.score)                  # 使用了这个装饰器，调用的时候就不用加括号了
```

```
de
姓名:de,成绩:33
33
```

普通继承：

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

    def show(self):
        info = '姓名:{},成绩:{}'.format(self.name,self.score)
        return info

class Dage(Student):
    def __init__(self,name,score,q_num):
        Student.__init__(self,name,score)      # 普通继承，使用Student.
        self.q_num = q_num
    def big_show(self):
        info = '{}考了{}分，有{}个枪。'.format(self.name,self.score,self.q_num)
        return info

ming = Dage('m',55,2)
print(ming.big_show())
print(ming.show())
```

```
m考了55分，有2个枪。
姓名:m,成绩:55
```

super（）继承：

1、https://www.runoob.com/w3cnote/python-super-detail-intro.html

2、https://blog.csdn.net/lb786984530/article/details/81192721

3、https://blog.csdn.net/zsh142537/article/details/82685721

```
class Parent():
  pass
class Son1(Paernt):
  super().__init__()
class Son2(Paernt):
  super().__init__()

class Grendson(Son1,Son2):
  super().__init__()

d = Grendson()
# 简单记录调用顺序：mro顺序
Grendson开始调用-->Son1开始调用-->Son2开始调用-->Parent开始调用-->Parent结束调用
-->Son2结束调用-->Son1结束调用-->Grendson结束调用
```

类中重构repr和str的思考和理解：

```
# 1/使用print,不带别的，调用__str__方法
class Demo():
    def __repr__(self):
        return 'sd'
    def __str__(self):
        return 'sss'

a = Demo()
print(a)            # 使用print时候，调用重构的str方法
print(str(a))       # 这句和上句实现的一模一样功能
```

```
sss
sss
```

```
# 2/使用repr()，调用__repr__方法
class Demo():
    def __repr__(self):
        return 'sd'
    def __str__(self):
        return 'sss'

a = Demo()
print(repr(a))    # 这样和下边一行输出一样的东西，调用重构的str方法
a
```

```
sd
sd
```

```
# 3/str比较随和，如果没有__str__,就调用__repr__
class Demo():
    def __repr__(self):
        return 'sd'
#     def __str__(self):
#         return 'sss'

a = Demo()
print(a)
```

```
sd
```

```
# 如果没有__repr__,不会去找__str__,而是向上层object里找
class Demo():
#     def __repr__(self):
#         return 'sd'
    def __str__(self):
        return 'sss'

a = Demo()
print(repr(a))
```

```
<__main__.Demo object at 0x109b40898>
```

*__new__*和*__init__*

> *__new__*是在`生成对象之前`所做的动作，接收参数是cls，由python解释器自动提供。
>
> *__init__*是在`对象生成之后`完善对象的属性，接收参数是self。
>
> 所以，*__new__*里要有return，才能生成对象，有了对象，init才开始工作。
>
> 绝大部分是不需要重写new的。

应用：new实现单例。只生成一个实例化对象。

```
class Singleton:
    instance = None
    def __new__(cls, xx, yy):
        if cls.instance is None:
            cls.instance = super().__new__(cls)
        return cls.instance
    def __init__(self, xx, yy):
        self.xx = xx
        self.yy = yy

obj1 = Singleton(1, 2)
obj2 = Singleton(2, 1)
print(obj1.xx, obj2.xx)
print(obj1 is obj2)
```

```
2 2
True
```

类中定义__call__方法，实现将类的实例化作为函数使用

> 类中如果定义了__call__方法，那么他的实例可以作为函数调用。

```
import random
class Bingocage:
```

```
    def __init__(self,item):
        self._item = list(item)  # 构建副本
        random.shuffle(self._item)

    def pick(self):
        return self._item

    def __call__(self):      # 定义__call__
        return self.pick()


bingo = Bingocage([1,2,3,4])
print(bingo.pick())           # 正常操作
print(bingo())                # 作为函数调用，即可以运用()运算符

print(callable(bingo))        # 判断是否可调用
```

```
[4, 2, 3, 1]
[4, 2, 3, 1]
True
```

流畅的Python—函数章节

> 内容提要：1、doc属性；2、函数的注解；3、函数带有默认参数；
>
> 3、operator模块的mul、itemmgetter；4、functools模块的partial、

函数的__doc__属性：

功能：打印三引号里的内容

```
def func():
    '''这是个函数的例子'''
    return 42
print(func.__doc__)
```

```
这是个函数的例子
```

函数注解：

注解不会做任何处理，只是存储在函数的__annotations__属性中。字典形式。

也就是说，注解对Python解释器没有任何意义。

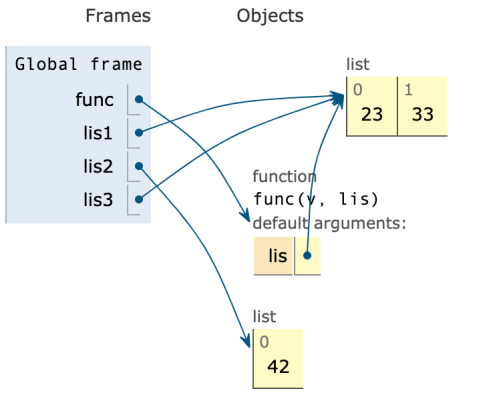
```
def clip(text,max_len=80):
# 对比    # 各个参数在：之后增加注解表达式，
# 如果参数有默认值，则把 注解 放在参数名和=之间
# 注解返回值，在最后的 ：  前添加  ->
def clip(text:str,max_len:'int>0'=80) -> str:
```

函数带有默认参数，但是默认参数只创建一次：

解释：lis1和lis3均使用默认lis[]，故指向相同；而lis2使用自己创建的[]。

```
1 def func(v,lis=[]):
2     lis.append(v)
3     return lis
4 lis1 = func(23)
5 lis2 = func(42,[])
6 lis3 = func(33)
7 print(lis1)
8 print(lis2)
9 print(lis3)
```

```
[23, 33]
[42]
[23, 33]
```



operator模块：

operator为多个运算符提供了对应的函数。

mul

```
from functools import reduce
from operator import mul              # 两数相乘,省的自己定义函数了

def func(n):
    return reduce(mul,range(1,n+1))

print(func(3))
```

```
6
```

itemgetter

```
from operator import itemgetter

lis = [('we','jp',33),('re','in',21),('ds','us',14),('cv','br',55)]
for i in sorted(lis,key=itemgetter(1)):   # 作用和lambda i:i[1] 一样，返回索引位1的元素
    print(i)
```

```
('cv', 'br', 55)
('re', 'in', 21)
('we', 'jp', 33)
('ds', 'us', 14)
```

attrgettr

(略)

functools模块：

partial冻结参数，即将原来的函数的一部分参数先给固定住

```
from operator import mul
from functools import partial

a = partial(mul,3)              # 把mul函数的第一个参数固定为3
print(a(8))
```

```
24
```

```python
# 接着上边的
list(map(a,range(5)))
```

```
[0, 3, 6, 9, 12]
```

lambda函数：

匿名函数：

除了作为参数传给高阶函数（如map）之外，Python很少使用lambda函数。

一道关于lambda函数的经典面试题:

> list[0]能输出什么？这个主要考函数对象列表，千万不要和列表表达式搞混了啊

```
list = [ lambda x:x*x for x in range(1, 3)]
print(list)
print(list[0])
print(list[1](5))
```

```
[<function <listcomp>.<lambda> at 0x7f3029c45ea0>, <function <listcomp>.
<lambda> at 0x7f3029c45e18>]
<function <listcomp>.<lambda> at 0x7f3029c45ea0>
25
```

题目：一行代码实现对列表a中的偶数位置的元素进行加3后求和?

```
a = [1,2,3,4,5]
sum(map(lambda y:y+3,filter(lambda x:a.index(x)%2==0,a)))
```

```
18
```

题目：对字典按value、key值排序：

！！sorted之后字典变成了元组构成的列表

```
#知识点：sorted函数。sorted()的参数，链接很详细。
# ！！sorted之后字典变成了元组组成的列表。
#https://www.runoob.com/python/python-func-sorted.html
dicc = {9149: 0, 9150: 26, 9151: 24, 9158: 24, 9153: 25}
dicc2 = sorted(dicc.items(),key=lambda m:m[1])  # 按value排序
dicc3 = sorted(dicc.items(),key=lambda m:m[0])  # 按key排序
print(dicc2,type(dicc2))
print(dicc3,type(dicc3))
```

```
[(9149, 0), (9151, 24), (9158, 24), (9153, 25), (9150, 26)] <class 'list'>
[(9149, 0), (9150, 26), (9151, 24), (9153, 25), (9158, 24)] <class 'list'>
```

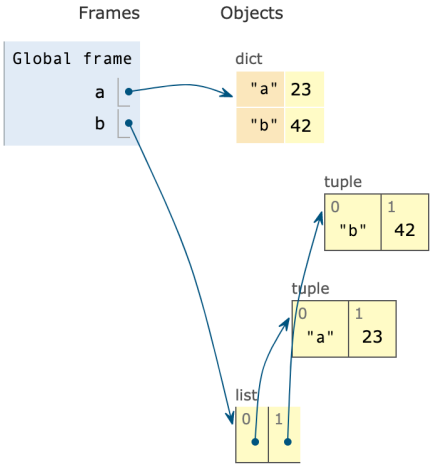
再看个例子：

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  a = {'a':23,'b':42}
2  b = sorted(a.items(),key=lambda m:m[1])
3  print(b,type(b))
4  print(b[0],type(b[0]))
```

Print output (drag lower right corner to resize)

```
[('a', 23), ('b', 42)] <class 'list'>
('a', 23) <class 'tuple'>
```



Python常用内置函数

> 内容提要：
>
> 1、bin()；2、all()；3、ord()和chr()；4、hasattr()；5、callable()；6、isinstance()；7、issubclass()；

bin函数：十进制转二进制

```
a = bin(15)        # 把15转成二进制
print(a,type(a))   # 输出字符串形式
```

```
0b1111 <class 'str'>  # 前边有0b
```

oct():十进制转八进制

hex():十进制转十六进制

int函数默认是转成十进制：

举个例子

```
print(int('34 '))      # 把字符串转成数字，会忽略空格
print(int('10',16))    # 把16进制的字符，转成十进制
```

all()函数：判断是否都为True

```
# all() 函数用于判断给定的可迭代参数 iterable 中的所有元素是否都为 TRUE，如果是返回 True，否则返回 False。

# 元素除了是 0、空、None、False 外都算 True。
```

ord函数和chr函数：ascii码和字符的转化

```
a = '!'
b = 97
print(ord(a))
print(chr(b))
```

```
33
a
```

hasattr()函数：判断对象是否包含对应的属性

> 如果对象有该属性返回 True，否则返回 False。
>
> 语法：hasattr(object, name) 参数：object -- 对象；name -- 字符串，属性名。

```
class Coor():
    x = 12
    y = 13
    z = 0
c = Coor()
print(hasattr(c,'x'))
print(hasattr(c,'bb'))
```

```
True
False
```

callable()函数：用来判断对象是否可调用的安全的方法

```
[callable(i) for i in (abs,str,13)]   # 数字不可调用
```

```
[True, True, False]
```

isinstance() :判断实例是否是某个类型

> 【Python3学习笔记p7】

```
isinstance(1.2,int)    # 1.2是否是int类型
```

```
False
```

2:同上

```
class Lion():
    pass
wxh = Lion()
isinstance(wxh,Lion)    # wxh是否是Lion的类型
```

```
True
```

```
isinstance(True,int)    # True是整型
```

```
True
```

issubclass():判断是不是某个类的子类

```
issubclass(Liger,Animal)  # 判断Liger是不是Animal的子类。注意顺序，谁是谁的子类
```

```
issubclass(bool,int)     # 证明了bool是int的子类
```

```
True
```

而object是所有类型的是共同祖先类

```
issubclass(int,object)
```

```
True
```

divmod()函数：

```
divmod(10,3)   # 输出商和余数的组合，tuple形式
```

```
(3,1)
```

Python内置函数整理

68个内置函数

> ### 分类记忆
>
> - 数学运算 × 7
>
> abs() 、 divmod() 、 max() 、 min() 、pow() 、round() 、sum()
>
> - 类型转换 × 24
>
> bool() 、 int() 、 float() 、 complex() 、str() 、 ord() 、 chr() 、 bytearray() 、 bytes() 、 memoryview() 、 bin() 、 oct() 、 hex() 、 tuple() 、 list() 、 dict() 、 set() 、 frozenset()、 enumerate() 、 range() 、 iter() 、 slice() 、super() 、object()
>
> - 序列操作 × 8
>
> all() 、any() 、 filter() 、 map() 、next() 、reversed() 、sorted() 、zip()
>
> - 对象操作 × 9
>
> help() 、dir() 、 id() 、hash() 、type() 、len() 、ascii() 、format() 、vars()
>
> - 反射操作 × 8
>
> Import() 、isinstance() 、issubclass() 、hasattr() 、getattr() 、setattr() 、delattr() 、callable()
>
> - 变量操作 × 2
>
> globals() 、locals()
>
> - 交互操作 × 2
>
> print()、input()
>
> - 文件操作 × 1
>
> open()
>
> - 编译执行 × 4
>
> compile() 、eval() 、exec() 、repr()
>
> - 装饰器 × 3
>
> property() 、 classmethod() 、staticmethod()

1、abs() 返回绝对值

```
abs(-30.4)
```

```
30.4
```

2、all() 所有元素都为真返回true，否则返回false

```
print(all([30,'d',0]))  # 含有非true元素
print(all(['dog',88]))
```

```
False
True
```

3、any()

```
print(any([30,'d',0]))  # 含有true元素
print(any([0]))         # 没有元素为真
print(any([]))          # 迭代器为空
```

```
True
False
False
```

4、bin() 将一个整数转变为一个前缀为“0b”的二进制字符串

```
bin(5)
```

```
'0b101'   # 字符串
```

5、chr()

```
chr(97)
```

```
'a'    # 97对应字母a
```

6、ord()

```
ord('a')
```

```
97
```

7、dict() 生成字典

```
dict(a=2,b=3)   # 注意！a,b没有引号
```

```
{'a': 2, 'b': 3}
```

8、dir() 返回一个列表，包含对象的所有属性

```
dir(int)
```

```
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 '__dir__',
...
]
```

9、divmod() 返回商和余数

```
divmod(11,5)   # 11除以5
```

```
(2, 1)
```

10、eval() 执行所给定的字符串表达式，并且返回表达式的值

```
eval('2*4')
```

```
8
```

11、hex() 返回整数对应的16进制

```
hex(13)
```

```
'0xd'
```

11.2、oct() 返回整数对应的8进制

```
oct(8)
```

```
'0o10'
```

12、id() 返回对象的内存地址

13、list() 传入的参数创建新的列表

```
list('3456')
```

```
['3', '4', '5', '6']
```

14、pow() 幂运算

```
pow(2,4)   # 2的4次方
```

```
16
```

15、

输出保留多少位：

1、

```
# %06d 整数输出,整数的宽度是6位,若不足6位,左边补0
x = 123
print('%06d'%x)
```

```
000123
```

2、

```
# %6d 整数输出,整数的宽度是6位,若不足6位,左边补空格
x = 123
print('%6d'%x)
```

```
   123
```

3、

```
# %-6d 整数输出,整数的宽度是6位,若不足6位,右you边补空格
x = 123
print('%-6d'%x)
```

```
123
```

4、

```
# %.6f 输出小数,即保留小数点后6位
```

库：collections.deque

> 内容提要：
>
> 1、右加，左加；2、清空；3、统计数目；4、右扩展，左扩展；5、元素索引；6、插入元素；7、右弹出，左弹出，指定删除；8、队列反转；9、rotate

```
from collections import deque

d = deque()
```

append（往右边添加一个元素）：

appendleft(往左边添加一个元素)：

```
d.append(1)
d.append(2)
d.appendleft(333)
```

清空：

```
d.clear()
```

✨count：返回指定元素的出现次数

```
d.count(1)  # d中，1出现的次数
```

extend：从队列右边扩展一个列表的元素

```
d.extend([2,3,5,7])
>>> d
>>> deque([1, 3, 4, 5])
```

extendleft：从队列左边扩展一个列表的元素

```
d.clear()
d.append(1)
d.extend([2,3,5,7])
>>> d
>>> deque([7,5,3,2,1])
```

index：查找某个元素的索引位置

```
>>> d
deque(['a', 'b', 'c', 'd', 'e'])
d.index("c",0,3) #指定查找区间,！前闭后开!
2
d.index("c",0,2)
ValueError: 'c' is not in deque
```

insert：在指定位置插入元素

```
>>> d
deque(['a', 'b', 'c', 'd', 'e'])
>>> d.insert(2,"z")
>>> d
deque(['a', 'b', 'z', 'c', 'd', 'e'])
```

pop（获取最右边一个元素，并在队列中删除）

popleft（获取最左边的一个元素，并在队列中删除）

remove：删除指定元素

```
>>> d
deque(['b', 'z', 'c', 'd'])
>>>
>>> d.remove("c")
>>> d
deque(['b', 'z', 'd'])
```

reverse：队列翻转

```
>>> d
deque(['a', 'b', 'c', 'd', 'c'])
>>> d.reverse()
>>> d
deque(['c', 'd', 'c', 'b', 'a'])
```

rotate：把右边元素放在左边

```
>>> d
deque(['c', 'd', 'c', 'b', 'a'])
>>> d.rotate(2)
>>> d
deque(['b', 'a', 'c', 'd', 'c']
```

heapq用法：

```
import heapq #载入heap库，heap指的是最小堆
```

1、将数组转换成堆：heapq.heapify(list)

```
lis = [1,3,4,2,6,8,9]
heapq.heapify(lis)
print(lis)
```

2、给堆增加元素，形成新堆：heapq.heappush(heap,item)

```
heapq.heappush(lis,5)
print(lis)
```

3、删除堆顶（即最小值）：heapq.heappop(heap)

```
heapq.heappop(lis)
print(lis)
```

4、删除最小值并添加新值，并形成新堆：heapq.heapreplace(heap, item)

```
heapq.heapreplace(lis,33)
print(lis)
```

5、查堆中的最大的N个数：heapq.nlargest (n, heap)

```
result = heapq.nlargest(2,lis)
print(result)
```

6、查堆中的最小的N个数：heapq.nsmallest(n, heap) #查询堆中的最小元素，n表示查询元素个数

```
result = heapq.nsmallest(2,lis)
print(result)
```

7、合并几个有序数组为一个有序数组。（和堆没啥关系。。）heapq.merge(a,b)

```
a = [1,2,3,4]
b = [6,8,9]
result = list(heapq.merge(a,b))
print(result)
```

Python常见概念：

> 内容提要：
>
> 1、设计模式；2、编码和解码；3、列表推导式和生成器；4、xrange和range；5、os.path和sys.path；
>
> 6、is和==；

设计模式

Python有三种设计模式：创建型、结构型、行为型。

创建型包括：`工厂模式`（要啥，都给你定义好）、`构造模式`（给你属性为空的类）、`原型模式`（克隆，再自己修改）、`单例模式`。

结构型包括：`代理模式`（如用列表实现栈）、`适配器模型`（鸭子类型）。

行为型包括：`迭代器模式`、`观察者模式`、`策略模式`（统一接口，内部使用不同的策略计算）。

编码和解码

- **python3默认编码为unicode，由str类型进行表示。**
- 二进制数据使用byte类型表示，所以不会将str和byte混在一起。
- 编码encode：把字符串变成byte类型。就是把好好地字符串编码绑架撕票剁成二进制。
- 解码decode：把byte类型变成字符串。
- 准确来说，Unicode不是编码格式，而是字符集。
- utf-8可以看成是unicode的一个扩展集。

列表推导式和生成器

列表推导式是将所有的值一次性加载到内存中，生成器并不创建一个列表，只是返回一个生成器。

xrange和range

相对来说，xrange比range性能优化很多，因为他不需要一下子开辟一块很大的内存，特别是数据量比较大的时候。

py3中，range就是2中的xrange.

os.path和sys.path

> os.path主要用于用户对系统路径的操作。
>
> sys.path主要用于python解释器的系统参数的操作。

is和==

> is通过id判断；==通过value判断。

+= 并不是原子操作，而是相当于extend和=两个动作。

字符串池：【P61】

相同的名字会出现在不同的名字空间里，就有必要实现共享。

因为内容相同，且不可变，所以共享不会导致任何问题。

池化能节约内存，且可省去创建新实例的开销。

Python就使用`字符串池`。

一旦失去所有的外部引用，池内的字符串一样会被回收。

深拷贝和浅拷贝

> 内容提要：1、可变类型；2、不可变类型；3、浅拷贝；4、深拷贝；5、栈内存和堆内存

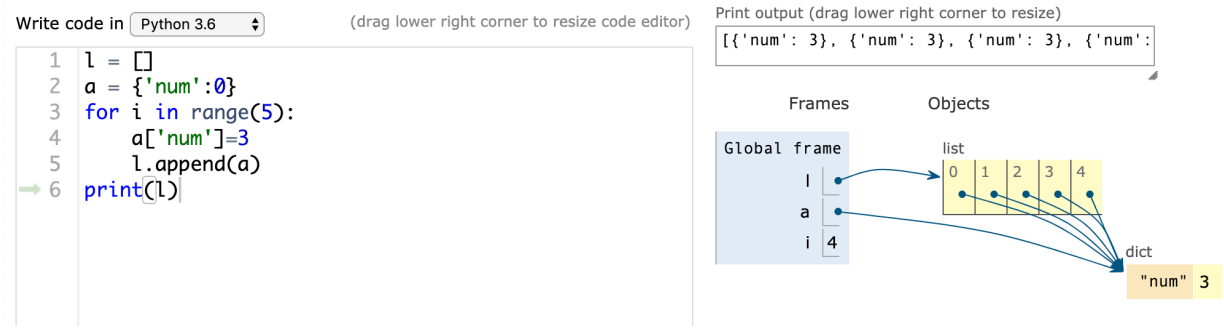
可变类型与不可变类型：

可变类型：列表，字典、可变集合

不可变类型：数字，字符串，元组、不可变集合(frozenset)

> 这里的可变不可变，是指内存中的那块内容（value）是否可以被改变

> 字典是可变对象，在下方的l.append(a)的操作中是把字典a 的引用传到列表l 中，当后续操作修改a['num']的值的时候，l 中的值也会跟着改变，相当于浅拷贝。



深拷贝和浅拷贝：

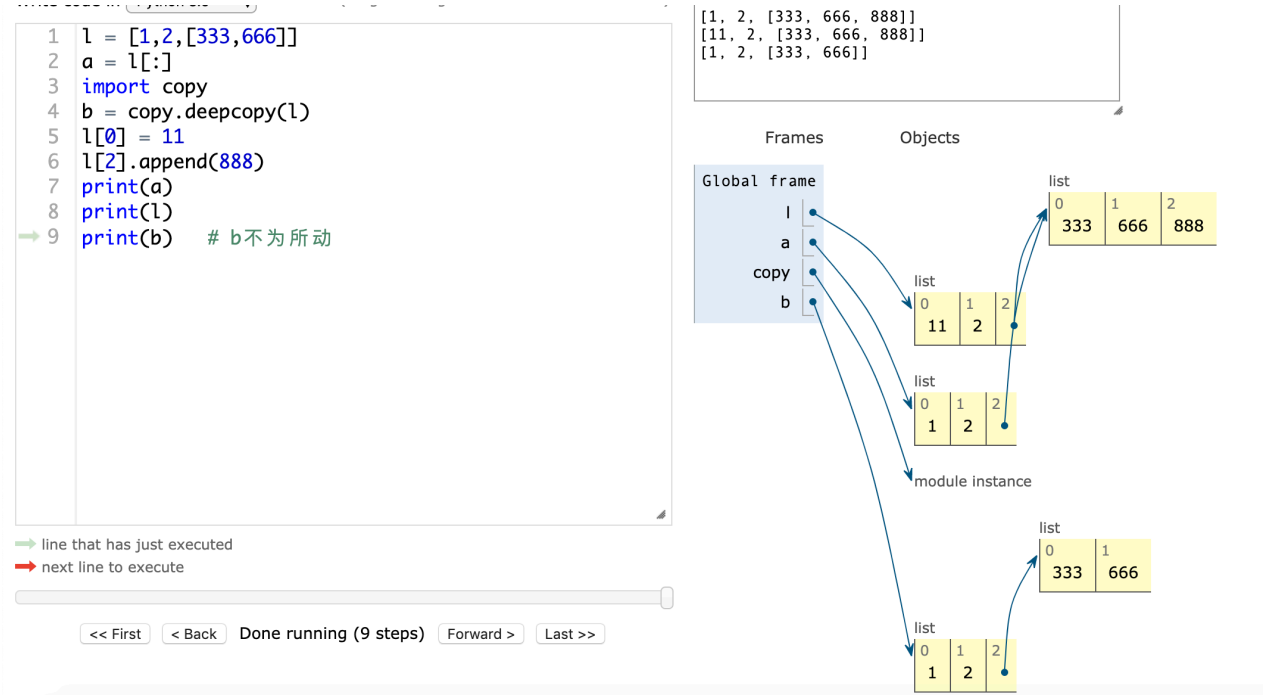
浅拷贝：3种形式

切片操作：`b = a[:] 或者b = [x for x in a];` copy 函数：`b = copy.copy(a);`工厂函数：`b = list(a);`

深拷贝只有1种形式：copy 模块中的`deepcopy()函数`。

a= l[:]是`浅拷贝`，只拷贝了第一层的，当l[2]改变时(即第二层改变)，a也会跟着改变。

b的赋值就是`深拷贝`,b不会随l而变化。



栈内存中存放的只是该对象的访问地址，在堆内存中为这个值分配空间。

由于这种值的大小不固定，因此不能把它们保存到栈内存中。

但内存地址大小的固定的，因此可以将内存地址保存在栈内存中。

这样，当查询引用类型的变量时， 先从栈中读取内存地址， 然后再通过地址找到堆中的值。对于这种，我们把它叫做按引用访问。

浅拷贝只复制指向某个对象的引用地址，而不复制对象本身，新旧对象还是共享同一块内存。

但深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象。

浅复制只复制一层对象的属性，而深复制则递归复制了所有层级。

装饰器：

> 内容提要：
>
> 1、什么是装饰器；2、装饰器实现单例模式；3、重新认识装饰器；4、functools.lru_cache装饰器；

什么是装饰器：

装饰器的目的是在不改变原函数名的情况下改变被包装对象的行为。

产生原因：已有的程序已经上线，不能大批修改源码。

举例：原函数是小盒子。装饰器是大盒子。使用中盒子来对小盒子做扩展功能。

大盒子返回中盒子的函数名，不带括号。

中盒子的定义体里包括小盒子，还包括其他功能。

看个例子就明白了。

使用装饰器实现单例模式：

```
def single01(cls):
    s = {}
    def wrap(*args,**kwargs):
        if 'j' not in s:
            s['j'] = cls(*args,**kwargs)
        return s['j']
    return wrap

@single01
class A():
    def __init__(self,name):
        self.name = name

a = A('du')
b = A('y')
print(a.name)
print(b.name)
```

```
du
du
```

重新认识装饰器:

```
@decorate
def target():
    print('hello')

# 上述代码的效果和下边的一样

def target():
    print('hello')
target = decorate(target)
```

> 装饰器有这么两个特点：1、把被装饰的函数替换成其他函数。2、**装饰器在加载模块时立即执行**。

关于第二点，看个例子：

```
registry = []
def register(func):                         # 定义装饰器
    print('我是装饰器，我在搞:%s'%func)
    registry.append(func)
    return func

@register                                   # 装饰f1
def f1():
    print('我是f1')

@register                                   # 装饰f2
def f2():
    print('我是f2')

def f3():                                   # 没有装饰f3
    print('我是f3')

print('registry里有这些：',registry)
f1()
f2()
f3()
```

```
我是装饰器，我在搞:<function f1 at 0x1054fc510>
我是装饰器，我在搞:<function f2 at 0x1054fc2f0>
registry里有这些：  [<function f1 at 0x1054fc510>, <function f2 at 0x1054fc2f0>]
我是f1
我是f2
我是f3
```

这个例子主要强调在加载模块时候立即执行，被装饰的函数（f1,f2,f3）只在明确调用时候才运行。此处显示了`导入时`和`运行时`之间的区别。

functools.lru_cache装饰器：

```
%%time
import functools
@functools.lru_cache()        # 有一对括号，原因是lru_cache可接受配置参数
def fib(n):
    if n<2:
        return n
    return fib(n-2) + fib(n-1)

print(fib(30))
```

```
832040
CPU times: user 266 µs, sys: 110 µs, total: 376 µs
Wall time: 328 µs
```

对比：

```
%%time
# import functools
# @functools.lru_cache()        # 有一对括号，原因是lru_cache可接受配置参数
def fib(n):
    if n<2:
        return n
    return fib(n-2) + fib(n-1)

print(fib(30))
```

```
832040
CPU times: user 572 ms, sys: 14.2 ms, total: 586 ms
Wall time: 781 ms
```

看见时间差异没？使用functools.lru_cache() 装饰器时，性能明显改善，因为他实现了缓存技术。

那现在具体看下他的参数配置：

`functools.lru_cache(maxsize=128,typed=False)`

maxsize指存储多少个调用结果，存满后，旧的会被扔掉，maxsize应设为2的幂。

typed如果设为True，通常会把1和1.0区分开。

另外，lru_cache使用字典存储，字典的键是根据电泳时传入的参数创建，因此，被装饰的函数的参数都必须是可以散列的。

functools.singledispatch装饰器：

抽空看帖子

Python多进程与多线程

> 内容提要：1.1、进程和线程的概念；1.2、多进程和多线程的区别；2、协程介绍及形象类比；3、多进程；4、进程池；5.1、进程的通信方式；5.2、队列实现进程间通信；6、多线程；7、进程和现成的效率和选择讨论

进程和线程概念：

进程是`资源分配`的最小单元，`独立内存`；线程是`操作系统调度`的最小单元，`共享内存`。

多线程和多进程的区别：

在单个程序中同时运行多个线程完成不同的工作，称为多线程；共享内存空间；同一个进程的线程之间可以直接交流。

多进程优点：稳定，一个子进程崩溃不会影响到其他子进程；

缺点：创建代价大，因为操作系统要给每个进程分配固定的资源。

多线程优点：效率高。

缺点：但是任何一个线程崩溃都可能造成整个进程的崩溃，因为多线程共享内存资源池。

CPU密集型使用多进程。IO密集型使用多线程。

单进程执行2次：

```
import time
import os


def long_time_task():
    print('当前进程：{}'.format(os.getpid()))   # 获取进程号
    time.sleep(2)
    print("结果：{}".format(8 ** 20))


if __name__ == "__main__":
    print('当前母进程：{}'.format(os.getpid()))
    start = time.time()
    for i in range(2):
        long_time_task()
    end = time.time()
    print("用时{}秒".format((end - start)))
```

```
当前母进程：15662
当前进程：15662
结果：1152921504606846976
当前进程：15662
结果：1152921504606846976
用时4.00621485710144秒
```

介绍下协程，为何比线程还快：

如果一个线程等待某些条件，可以充分利用这个时间去做其它事情，其实这就是：协程方式。

协程创建的开销与线程相比完全可以忽略，这意味着，使用多协程可以处理更多的任务。原因是协程能保留上一次调用的状态。

拓展：

如果某个员工在上班时临时没事或者再等待某些条件（比如等待另一个工人生产完谋道工序 之后他才能再次工作），那么这个员工就利用这个时间去做其它的事情，那么也就是说：如果一个线程等待某些条件，可以充分利用这个时间去做其它事情，其实这就是：协程方式

Python多进程：multiprocessing模块的Process

> 注释：实际运行中包含了1个母进程和2个子进程

```
from multiprocessing import Process
import os
import time


def long_time_task(i):
    print('子进程: {} - 任务{}'.format(os.getpid(), i))
    time.sleep(2)
    print("结果: {}".format(8 ** 20))


if __name__ == '__main__':
    print('当前母进程: {}'.format(os.getpid()))
    start = time.time()
    p1 = Process(target=long_time_task, args=(1, ))    # 两个参数：函数名和向函数传
的参数
    p2 = Process(target=long_time_task, args=(2, ))
    print('等待所有子进程完成。')
    p1.start()                                         # 启动进程
    p2.start()
    p1.join()                                          # 阻塞母进程，等待子进程结束
    p2.join()
    end = time.time()
    print("总共用时{}秒".format((end - start)))
```

```
当前母进程: 15662
等待所有子进程完成。
子进程: 18152 - 任务1
子进程: 18153 - 任务2
结果: 1152921504606846976
结果: 1152921504606846976
总共用时2.0279181003570557秒
```

Python多进程：multiprocessing模块的Pool

```
from multiprocessing import Pool, cpu_count
import os
import time


def long_time_task(i):
    print('子进程: {} - 任务{}'.format(os.getpid(), i))
    time.sleep(2)
    print("结果: {}".format(8 ** 20))


if __name__ == '__main__':
    print("CPU内核数:{}".format(cpu_count()))   # 打印cpu的核数
    print('当前母进程: {}'.format(os.getpid()))
    start = time.time()
    p = Pool(4)                         # 开启容量为4的进程池
    for i in range(5):
        p.apply_async(long_time_task, args=(i, ))
    print('等待所有子进程完成。')
    p.close()                           # 关闭进程池，使不在接受新的任务
    p.join()                            # 阻塞主进程，join()要用在close()或者
terminate()之后
    end = time.time()
    print("总共用时{}秒".format((end - start)))
```

```
CPU内核数:4
当前母进程: 15662
子进程: 18181 - 任务0
子进程: 18183 - 任务2
子进程: 18182 - 任务1
子进程: 18184 - 任务3
等待所有子进程完成。
结果: 1152921504606846976
结果: 1152921504606846976
结果: 1152921504606846976
子进程: 18183 - 任务4
结果: 1152921504606846976
结果: 1152921504606846976
总共用时4.12725305557251秒
```

进程通信的方式：

管道、消息队列、共享内存（不同虚拟地址和同一物理地址的映射）、信号量。

使用队列实现进程间通信：Queue

```
from multiprocessing import Process, Queue
import os, time, random


def write(q):       # 写数据进程
    print('Process to write: {}'.format(os.getpid()))
    for value in ['A', 'B', 'C']:
        print('Put %s to queue...' % value)
        q.put(value)
        time.sleep(random.random())

def read(q):        # 读数据进程
    print('Process to read:{}'.format(os.getpid()))
    while True:
        value = q.get(True)
        print('Get %s from queue.' % value)


if __name__=='__main__':
    q = Queue()     # 父进程创建Queue，并传给各个子进程
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    pw.start()
    pr.start()
    pw.join()       # 等待pw结束
    pr.terminate()  # pr进程里是死循环，无法等待其结束，只能强行终止
```

```
Process to write: 18479
Put A to queue...
Process to read:18480
Get A from queue.
Put B to queue...
Get B from queue.
Put C to queue...
Get C from queue.
```

Python多线程

threading模块的Thread：

> 可以看到也是只用了2.几秒，GIL锁没有锁子线程么？
>
> 有的。但是睡眠时候，并没有占用cpu，就释放GIL锁了。

```
from threading import Thread,current_thread
import os
import time


def long_time_task(i):
    print('子进程: {} - 任务{}'.format(current_thread().name,i)) # 打印主线程的名字
    time.sleep(2)
    print("结果: {}".format(8 ** 20))


if __name__ == '__main__':
    print('当前主线程: {}'.format(current_thread().name))
    start = time.time()
    t1 = Thread(target=long_time_task, args=(1, ))    # 两个参数：函数名和向函数传的
参数
    t2 = Thread(target=long_time_task, args=(2, ))
    print('等待所有子进程完成。')
    t1.start()                                          # 启动进程
    t2.start()
    t1.join()                                           # 阻塞母进程，等待子进程结束
    t2.join()
    end = time.time()
    print("总共用时{}秒".format((end - start)))
```

```
当前主线程: MainThread
等待所有子进程完成。
子进程: Thread-14 - 任务1
子进程: Thread-15 - 任务2
结果: 1152921504606846976结果: 1152921504606846976

总共用时2.0085318088531494秒
```

多进程和多线程的效率问题：

- 对CPU密集型（如循环） —>多进程效率更高
- 对IO密集型（如文件操作、爬虫） —>多线程效率更高

因为对于IO密集型，大部分时间是花在等待，等待时间是不占用CPU的，Python遇到等待会释放GIL供新的线程使用，实现了线程的切换。

变量的作用域和闭包

变量作用域：

先看个例子，吃上一惊。回头再解释。

```
b = 8
def f2(a):
    print(a)
    print(b)
    b = 9       # b在此处定义
f2(6)
```

```
6
UnboundLocalError: local variable 'b' referenced before assignment
```

报错看见没，上边有全局变量b啊，为啥还报错。（先说个没写出来的，如果b=9注释掉，结果是6 8）。

下面解释报错的原因：

> Python在编译函数的定义体时候，他判断b是局部变量，生成的`字节码`证实了这种判断，python会尝试从本地环境获取b,但是打印b时候b还没绑定值。所以就报错了。
>
> Python不要求声明局部变量，但是假定在函数体中的变量是局部变量。

LEGB:

闭包：

```
def func():
    liss = []
    def f(a):
        liss.append(a)          # 在函数f中，liss是自由变量，指没在本地作用域中绑定的变量
        total = sum(liss)
        return total/len(liss)
    return f

aa = func()
print(aa(2))                    # 调用函数func()
print(aa(8))
```

```
2.0
5.0
```

注意，在调用函数aa(2)时候，func()函数就已经返回了，他的本地作用域也拜拜了。

> 但是闭包会保留定义函数时候存在的自由变量的绑定，这样的话，在调用函数时，虽然func()的作用域拜拜了，但是仍然能使用那些绑定。

下边来看一个错误的例子：

```
def func():
    count = 0
    total = 0
    def f(a):
        count += 1
        total += a
        return total/count

aa = func()
aa(2)
```

```
报错:UnboundLocalError: local variable 'count' referenced before assignment
```

原因：首先count是不可变类型。count += 1的操作相当于在f的函数体中对count赋值了，**这就隐式的把count变成了局部变量，那就不是自由变量了**，就不能保存在闭包中了。total也是一样的。

那怎么办呢？看注释。

```
def func():
    count = 0
    total = 0
    def f(a):
        nonlocal count,total        # 使用nonlocal声明成全局变量
        count += 1
        total += a
        return total/count
    return f

aa = func()
print(aa(2))
print(aa(8))
```

```
2.0
5.0
```

Python的垃圾回收：

> 内容提要：
>
> 1、介绍；2、循环引用垃圾回收；

介绍下垃圾回收机制

引用计数为主，分代回收和标记为辅。

引用计数、分代回收（0，1，2这三代）（对象存在的时间越长，越不可能是垃圾）、标记回收（标记阶段、清除阶段）、孤立的引用环。

关于循环引用垃圾回收的补充：

```
class X:
    def __del__(self):
        print(self,'dead.')

a = X()
b = X()
a.x = b
b.x = a

import gc               # 使用循环引用的垃圾回收
del a
del b
gc.enable()             # 开启
gc.collect()            # 主动启动回收操作，循环引用对象被正确回收
```

```
<__main__.X object at 0x1085246a0> dead.
<__main__.X object at 0x108524cc0> dead.
```

弱引用：

（终于等到你）但是我觉得这段讲的不好

> 名字和对象之间关联成强引用关系，也就是说会增加引用计数。那么弱引用就是在保留引用的前提下，不增加引用计数，亦不阻止对象的回收。

```
class X():
    def __del__(self):
        print(id(self),"dead.")

a = X()
print(sys.getrefcount(a))    # 打印a的引用计数

import weakref
w = weakref.ref(a)           # 创建弱引用
print(w() is a)              # 通过弱引用访问对象。注意w后跟括号
print(sys.getrefcount(a))    # 引用计数没变化

print(w())
print(a)

del a
print(w() is None)           # 返回True
```

```
2
True
2
<__main__.X object at 0x108413470>
<__main__.X object at 0x108413470>
4433458288 dead.
True
```

留个尾巴，并不是所有数据类型都支持弱引用。如int，tuple

弱引用的相关函数：

```
a = X()
w = weakref.ref(a)
weakref.getweakrefcount(a)    # 对象a的弱引用计数
```

```
1
```

那，可以看到，w作为弱引用，在调用的时候要使用w().

有什么办法，让我只使用w就能直接调用这个弱引用呢?

用proxy

```
a = X()
a.name = 'Qee'
w = weakref.ref(a)       # 没使用代理，下行需要用()
print(w().name)
p = weakref.proxy(a)     # 使用代理，调用不需要用()
print(p.name)
```

```
Qee
Qee
```

正则：

1.直接给出字符，就是精确匹配。用`\d`可以匹配一个数字，`\w`可以匹配一个字母或数字

2.`.`可以匹配任意字符

3.用`*`表示任意个字符（包括0个）

用`+`表示至少一个字符，

用`?`表示0个或1个字符

用`{n}`表示n个字符，

用`{n,m}`表示n-m个字符

4.`-`这样的是特殊字符，在正则表达式中，要用`\`转义。

4.`\s`可以匹配一个空格（也包括Tab等空白符），所以`\s+`表示至少有一个空格，例如匹配' '，' '等

5.`[0-9a-zA-Z\_]`可以匹配一个数字、字母或者下划线。**匹配的是一个。**

`[0-9a-zA-Z\_]+`可以匹配至少由一个数字、字母或者下划线组成的字符串，比如'a100'，'0_Z'，'Py3000'等等；

6.`A|B`可以匹配A或B，所以`(P|p)ython`可以匹配'Python'或者'python'

7.`^`表示行的开头，`^\d`表示必须以数字开头。`$`表示行的结束，`\d$`表示必须以数字结束。

8.强烈建议使用Python的r前缀，就不用考虑转义的问题了

9.正则表达式还有提取子串的强大功能。用`()`表示的就是要提取的分组（Group）。比如：

`^(\d{3})-(\d{3,8})$`分别定义了两个组

10.正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。由于`\d+`采用贪婪匹配，直接把后面的`0`全部匹配了，结果`0*`只能匹配空字符串了。

必须让`\d+`采用非贪婪匹配（也就是尽可能少匹配），才能把后面的`0`匹配出来，加个`?`就可以让`\d+`采用非贪婪匹配：

```
>>> re.match(r'^(\d+?)(0*)$', '102300').groups()
('1023', '00')
```

11.当我们在Python中使用正则表达式时，re模块内部会干两件事情：

1)编译正则表达式，如果正则表达式的字符串本身不合法，会报错；

2)用编译后的正则表达式去匹配字符串。

如果一个正则表达式要重复使用几千次，出于效率的考虑，我们可以预编译该正则表达式，接下来重复使用时就不需要编译这个步骤了，直接匹配：

```
>>> import re
# 编译:
>>> re_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')
# 使用:
>>> re_telephone.match('010-12345').groups()
('010', '12345')
>>> re_telephone.match('010-8086').groups()
('010', '8086')
```

实操题目：

> 内容提要：
>
> 0、判断数字是奇数还偶数、输出随机数、magic函数；1、把IP地址转化成32位二进制；2、统计文章；

判断数字是奇数还是偶数：推荐

```
# 和1进行与运算   n&1
# 与运算：        全1才出1

# 如果n&1返回0，那么n为偶数；
# 否则返回1，n为奇数。
```

输出随机数:

```
import random
print(random.randint(4,10))  # 输出4，10之间的随机整数
print(random.uniform(4,10))  #  输出4，10之间的随机数
```

magic函数:

在编程的时候有时候我们想要比较 两种算法哪个更快 或者自己的代码哪一段最慢 这时候就可以使用magic函数

> magic有行魔法%time 和单元魔法%%time 行魔法 显示这一行代码运行的时长 单元魔法显示这一个cell 运行的时长

题目：把IP地址转化成32位二进制：

```
a = '127.0.0.1'
b = list(map(int,a.split('.')))
c = ''
for i in b:
    tem = bin(i).replace('0b','')
    while len(tem)<8:
        tem = '0'+tem
    c = c + tem
print(c)
print(len(c))
```

```
01111111000000000000000000000001
32
```

题目：用python实现统计一篇英文文章内每个单词的出现频率，并返回出现频率最高的前10个单词及其出现次数。

```
import re

with open('this.txt','r') as f:
    word_list = []
    word_dict = {}
    for line in f.readlines():
        for word in line.strip().split():
            word_list.append(re.sub('[,|.|!|*|-]','',word.lower()))
    word_set = list(set(word_list))
    word_dict = {word:word_list.count(word) for word in word_set}
result = sorted(word_dict.items(),key = lambda i:i[1], reverse=True)[:10]
print(len(result))
print(result)
```

```
import re
from collections import Counter

with open('this.txt','r') as f:
    text = f.read()
    count = Counter(re.split('\W+',text))

result = count.most_common(10)
print(result)
print(count)
```

题目：长度为n的无序列表，求其中位数

思路：取前一半元素，构成最小堆。遍历剩下的一半，比堆顶小则不考虑；比堆顶大则替换堆顶，重新构成最小堆。

解释：最后的这个堆顶元素，比那些不考虑的元素大，比堆中其余元素小，可不就是中位数么。

题目：斐波那契数列?

```
def fib (num):
    numlist = [0,1]
    for i in range(num-2):
        numlist.append(numlist[-2]+numlist[-1])
    return numlist
```

题目：进制转换：十进制转二进制

```
# divmod(a, b)函数:输出a/b的商和余数。输出元组形式
def ten_to_two(num):
    lis = []
    if num == 0:
      return 0
    while num:
        num,rem = divmod(num,2)    # divmod()函数
        lis.append(str(rem))       # 转换成字符串
    return ''.join(reversed(lis)) # 字符串拼接

# 测试
ten_to_two(8)
```

使用堆合并K个有序链表：

```
from heapq import heappop, heapify
class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution(object):
    def mergeKLists(self, lists):
        ## :type lists: List[ListNode]   :rtype: ListNode
        # 1、合并成最小堆
        h = []
        for lis in lists:   # lists是链表
            while lis:
                h.append(lis.val)
                lis = lis.next
        heapify(h)

        if not h:           # 防止lists为[]或者[[]]
            return None
        # 2、构造链表
        root = ListNode(heappop(h))
        curnode = root
        while h:
            nextnode = ListNode(heappop(h))
            curnode.next,curnode = nextnode,nextnode
        return root
```