

提示：涵盖Python基础语法，掌握基础的编程能力

## 01、python初识

- 1.1、掌握Python注释
- 1.2、掌握Python变量的定义与命名规则
- 1.3、掌握Python基本操作符
- 1.4、掌握Python输入输出

## 02、Python流程控制

- 2.1、单分支
- 2.2、双分支
- 2.3、多分支
- 2.4、if-else 嵌套
- 2.5、while循环
- 2.6、for循环
- 2.7、案例体现

## 03、Python数据类型

- 字符串
- 列表
- 元组
- 字典
- 共有方法
- 不可变序列与可变序列
- set集合
- 总结

## 04、Python函数

- 4.1、函数初识
- 4.2、函数参数
- 4.3、函数返回值
- 4.4、名称空间与作用域
- 4.5、函数嵌套
- 4.6、函数全局变量
- 4.7、引用
- 4.8、函数对象和闭包
- 4.9、装饰器
- 4.10、迭代器
- 4.11、生成器
  - yield表达式（挂起）
  - 三元表达式
  - 列表生成式
  - 集合生成式
  - 生成器表达式
- 4.12、二分法
- 4.13、面向过程与函数式
  - 面向过程
  - 函数式
    - 匿名函数lambda
    - map、filter、reduce（了解）
- 4.13、函数的递归调用
- 4.14、内置函数 数学运算
- 4.115、类型转换函数
- 4.16、序列操作函数

## 05模块与包

- 5.1、首次导入模块发生的三件事

- 5.2、import导入模块使用
- 5.3、from ...import 导入
- 5.4、循环导入问题
- 5.5、模块搜索优先级
- 5.6、代码规范
- 5.7、函数的提示类型
- 5.8、包

## 06、Python面向对象

- 6.1、oop介绍
- 6.2、类和对象
- 6.3、定义类
- 6.4、**init**方法
- 6.5、属性访问
  - 类属性与对象属性
- 6.5、self理解
- 6.6、魔术方法
- 6.7、析构方法
- 6.8、封装
- 6.9、继承
  - 单继承
  - 多继承
  - 属性查找
  - 继承实现原理
- 6.10、重写父类方法
- 6.11、组合
- 6.12、Mixins机制
- 6.13、多态
- 6.14、类属性和实例属性
- 6.15、类方法和静态方法
- 6.16、私有化属性(隐藏)
- 6.17、私有化方法(隐藏)
- 6.18、Property函数
- 6.19、`__new__` 方法
- 6.20、单例模式
- 6.21、异常处理
- 6.22、反射机制
- 6.23、元类

## 07、文件操作

- 7.1、文件操作流程
- 7.2、文件的操作模式
- 7.3、操作文件的方法
- 7.4、主动移动文件内指针移动
- 7.5文件的修改
- 7.6垃圾回收机制

## 08、正则表达式

- 8.1、限定匹配数量规则
- 8.2、原生字符串
- 8.3、分组匹配
- 8.4、编译函数compile
- 8.5贪婪模式和非贪婪模式

# 01、python初识

---

Python是一种面向对象的解释型计算机程序设计语言，由吉多.范罗苏姆开发，第一个公开发行人版发布于1991年，它常被成为胶水语言，能够把其他语言制作的各种模块（尤其是C/C++），恒轻松的联结在一起。

## Python优点

- 免费开源  
Python开源，开发者可以自由的下载，阅读，甚至是修改Python源代码
- 丰富的第三方库  
Python具有本身且丰富而且强大的库，而且由于Python的开源性，第三方库也非常多，例如：在web开发有django.flask,Tornado，爬虫scrapy，科学计算numpy.pandas等等
- 可以移植  
由于Python是开源的，他已经被移植到了大多数平台下面，例如：Windows，MacOS，Linux，Andorid,iOS等等
- 面向对象  
Python既支持面向过程，又支持面向对象，这样编程就更加灵活

## Python缺点

- 运行速度慢  
C程序相比非常慢，因为Python是解释型语言，代码在执行时会一行一行的翻译成CPU能够理解的机器码，这个翻译过程非常的耗时，所以很慢，而C程序时运行前直接编译成CPU能执行的机器码，所以相对Python而言，C语言执行非常快
- 代码不能加密  
要发布你写的程序，实际上时发布源代码，而是解释型的语言，则必须把源代码发布出去
- 强制的缩进  
Python有非常**严格的缩进语法**，只要缩进错误程序立马崩溃
- GIL全局解释器锁  
在任意时刻，只有一个线程在解释器中运行，对Python虚拟机的访问由全局解释器锁（GIL）来控制，正式这个锁能够保证同一时刻只有一个线程在运行，遇到i/o阻塞的时候会释放（GIL）所以Python的多线程并不是真正的多线程，而是CPU执行速度非常快，让人感觉不到GIL的存在。  
（GIL）会在Python高级阶段讲解

## 1.1、掌握Python注释

- 注释是编写程序时，写程序的人给人一句、程序段、函数等的**解释或提示**
- 注释的作用：  
注释可以起到一个备注的作用，这个方法函数，变量到底是干嘛的，如果没有注释时间长了即使是自己写的代码，可能都不知道这个代码到底是干嘛的，所以注释起到的作用就是方便自己查看写过的代码，别人来接收你的代码能看懂，简单来将就是提高程序代码的可读性，以便于以后的参看、修改
- Python中**单行注释用#号**，#号右边就是注释的内容，Python解析器遇到#号就会当作注释，不会去解析#号后面的内容

### 多行注释

```
1  """
2  多行注释
3  """
```

## 1.2、掌握Python变量的定义与命名规则

- 变量=存储的数据
- 变量是一段有名字的连续的存储的空间，我们可以通过定义变量来申请并命名这样的存储空间，并通过变量的名字来使用这段存储空间
- 变量是程序中临时存放数据的场所  
Python是一种强类型的语言，赋值变量时候不需要指定数据类型，给这个变量赋值什么数据类型，这个变量就是什么类型

```
1 #x就是变量，当x=2结果就是6 x=10结果就是30
2 x=2
3 y=x*3
4 print(y)
```

### 命名规则：

变量必须以字母（a-z A-Z）下划线（\_）开头，其他字符可以是字母数字下划线，变量区分大小写，Python关键字不能作变量名

```
1 _Name='张三'
2 name='刘德华'
3 print(_Name,name)
```

### 命名规范：

- 简明知意，尽量使用有语义的单词命名，如使用Password用作密码，username用作用户名
- 小驼峰式命名法：第一个单词首字母小写其他单词首字母大写如userName
- 大驼峰式命名法：全部单词首字母都用大写，如UserName
- 下划线命名法：每个单词用\_下划线连接，如user\_name

## 1.3、掌握Python基本操作符

- 算数运算符：+ - \* / \*\*（指数）%（取余）//（相除取整）

定两个变量a=7,y=3

算数运算符	作用描述	示例
+ 加法	算术加法	a+b=10
-减法	算数减法	a-b=4
*乘法	算数乘法	x*y=21
**指数	左边的数是底数，右边的数是指数	a* *b=343
%取余	x%y x除以y的余数	a%b=1
/除法	x/y结果包含小数点后面的数	a/b=2.33333333335
//相除取整	x//y 结果是忽略小数点后面的小数位，只保留整数位	a//b=2

- 比较运算符：==（等于）!=（不等于）> < >= <=

返回布尔型 True False

比较运算符	名称	示例	结果描述
==	等于	x==y	如果x恰好等于y，则为真
!=	不等于	x!=	如果x恰好不等于y 则为真
>	大于	x>y	如果x(左侧参数)大于y(右边参数),则为真
<	小于	x< y	如果x(左侧参数)小于y(右边参数),则为真
>=	大于或等于	x>=y	如果x(左侧参数)大于或者等于y(右边参数),则为真
<=	小于或者等于	x<=y	如果x(左侧参数)小于或者等于y(右边参数),则为真

- 逻辑运算符 and or not

逻辑运算符	实例	结果描述
and	x and y	x,y同为真，则结果为真，如果有一个为假，则结果为假
or	x or y	x, y有一个为真，结果就为真，全部为假，则结果为假
not	not x	取反，如果x为真，则结果为假，如果x为假，则结果为真

```
1 a=1
2 b=2
3 c=3
4 d=5
5 print(a>b and d>c) #结果为假
6 print(a>b or d>c) #结果为真
7 print(not a>d) #结果为真
```

优先级：（）-->not-->and-->or

```
1 print(2>1 and 1<4 or 2<3 and 9>6 or 2<4 and 3<2) #True
```

赋值运算符	作用描述	结果描述
=	赋值运算符	将=号右边的值赋值给左边的变量
+=	加法赋值运算符	c+=a等效于c=c+a
-=	减法赋值运算符	c-=a等效于c=c-a
*=	乘法赋值运算符	c*=a等效于c=ca
/=	除法赋值运算符	c/=a等效于c=c/a
%=	取模赋值运算符	c%=a等效于c=c%a
**=	幂赋值运算符	c*=a等效于c=c *a
//=	取整赋值运算符	c//=a等效于c=c//a

```

1 a,b,c,d=23,10,18,3
2 a+=3
3 print(a)

```

## 1.4、掌握Python输入输出

### 输出 %占位符

- Python 有一个简单的字符格式化方法，使用%做占位符，%后面跟的是变量的类型 %s %d
- 在输出时，若果有\n,此时\n后的内容会在另外一行显示 换行

```

1 name ='张三'
2 classPro='北京大学'
3 age =22
4 print('我的名字是%s: 来自%s 今年%d岁了'%(name,classPro,age))
5
6 me ='我的'
7 classP='清华附中一年级3班'
8 age1=7
9 print('%s名字是小明，来自%s,我今年%d岁了'%(me,classP,age1))
10
11 print('我可以\n换行吗')
12

```

- %c 字符
- %s 通过str()字符串转换来格式化
- %d 有符号十进制整数
- %u 无符号十进制整数
- %f浮点数

```

1 print('\n')
2
3 print('-----test-----')
4 print('\n')
5 ID='老夫子'
6 QQ=66666666
7 number=13467204515
8 address='广州市白云区'
9 print('=====')
10 print('姓名: %s'%ID)
11 print('QQ: %d'%QQ)
12 print('手机号码: %d'%number)
13 print('公司地址: %s'%address)
14 print('=====')
15 print('\n')

```

### 格式化输出另外一种方式

```
1 print('-----格式化输出另外一种方式.format-----')
2 print('姓名: {}'.format(ID))
3 print('QQ: {}'.format(QQ))
4 print('手机号码: {}'.format(number))
5 print('公司地址: {}'.format(address))
```

## 输入 input

Python中提供input 方法来获取键盘输入

**input接收的键盘输入都是str类型**，如果接收数字类型需要将str转成int

```
1 name=input('请输入姓名')
2 age2=int(input('请输入年龄: '))
3 qq=input('请输入QQNumber: ')
4 phone=input('请输入手机号: ')
5 addr=input('请输入地址: ')
6 print('姓名: %s 年龄是%d'%(name,age2))
7 print('QQ: {}'.format(qq))
8 print('手机号码: {}'.format(phone))
9 print('公司地址: {}'.format(addr))
10
```

# 02、Python流程控制

- Python条件语句是通过一条或多条语句的执行结果【True/False】来决定执行的代码块
- 就是**计算机执行代码的顺序**
- 流程控制：对计算机代码执行的顺序进行有效的管理，只有流程控制才能实现在开发当中的业务逻辑
- **流程控制的分类：**
  - 顺序流程 ：就是代码一种自上而下的执行结构，也是python默认的流程
  - 选择流程/分支流程：根据在某一步的判断，有选择的去执行相应的逻辑的一种结构
    - 单分支 if 条件表达式： if
    - 双分支 if 条件表达式： if else:
    - 多分支 if 条件表达式： if elif 条件表达式： elif 条件式： else:
    - **条件表达式**:比较运算符/逻辑运算符/符合运算符
  - 循环流程： 在一定的条件下，一直重复的去执行某段代码的逻辑【事情】
    - while 条件表达式:
    - for 变量 in 可迭代的集合对象

## 2.1、单分支

```
1 score = 80
2 if score<=60: #满足条件就会输出打印提示
3     print('成绩不太理想，继续加油')
4     pass #空语句
5 print('语句运行结束')
```

## 2.2、双分支

```
1 if score>60:
2     print('成绩不错')
3 else:
4     print("成绩不合格，继续努力")
```

## 2.3、多分支

- 特征：只要满足其中一个分支，就会退出本层if 语句结构【必定会执行其中一个分支】
- 至少有2种情况可以选择
- elif后面必须得写上条件和语句
- else 是选配，根据实际情况来填写

```
1 grade=int(input('请输入成绩：'))
2 if grade>90:
3     print('您的成绩是A等')
4     pass
5 elif grade>=80:
6     print('您的成绩是B等')
7     pass
8 elif grade >=70:
9     print('您的等级是C等')
10    pass
11 elif grade >=60:
12    print('您的等级是D等')
13    pass
14 else:
15    print('你个废物蛋')
16 print('程序运行结束.....')
17
```

## 2.4、if-else 嵌套

- 一个场景需要分阶段或者层次，做出不同的处理
- 要执行的if语句一定要外部的if 语句满足条件才可以，

```
1 xuefen=int(input('请输入你的学分'))
2 if xuefen >10:
3     grade = int(input('请输入你的成绩'))
4     if grade>=80:
5         print('你可以升班了。。。恭喜您')
6         pass
7     else:
8         print('很遗憾，您的成绩不达标....')
9         pass
10    pass
11 else:
12    print('您的表现也太差了吧0')
```



## 2.5、while循环

语法特点

1.有初始值

2.条件表达式

3.变量【循环体内计数变量】的自增自减，否则会造成死循环

- 使用条件：循环的次数不确定，是依靠循环条件来结束
- 目的：为了将相似或者相同的代码操作变得更加简洁，使得代码可以使用重复利用

## 2.6、for循环

- 遍历操作，依次的取集合容器中的每个值

```
1  tage='我是一个中国人'#字符串类型本身就是一个字符类型的集合
2  for item in tage:
3      print(item)
4      pass
5  #range 此函数可以生成一个数据集合的列表
6  #range(起始值: 结束值: 步长) 步长不能为0
7  sum=0
8  for data in range(1,101):#数据做包含，右边不包含1-100
9      sum+=data
10     pass
11     #print(data,end=' ')
12 print(sum)
13 print('sum=%d'%sum)
14 # print('sum={}'.format(sum))
15 #
16 for message in range(50,201):
17     if message%2==0:
18         print('%d是偶数'%message)
19         pass
20     else:
21         print('{}是奇数'.format(message))
22
23 # break 代表中断结束，满足条件直接结束本层循环
24 # continue 结束本次循环，继续进行下次循环，（当continue条件满足的时候，本次循环剩下的
    语句将不在执行，后面的循环继续）
25 #这两个关键字只能用在循环中
```

**while:**适用于对未知的循环次数【用于判断】

**for :** 适应于已知的循环次数【可迭代对象遍历】

## 2.7、案例体现

99乘法表用for实现

```

1  #九九乘法表
2  for i in range(1,10):
3      for j in range(1,i+1):
4          print('%d*%d=%d'%(i,j,i*j),end=' ')
5          print()#控制换行
6
7  print('-----')
8  print('\n')

```

### 案例 输入1-100之间的数据

```

1  #输入1-100之间的数据
2  index =1#定义变量
3  while index<=100:
4      print(index)
5      index+=1
6      pass

```

### 案例猜拳游戏

```

1  #案例猜拳游戏
2  import random
3  count =1
4  while count<=10:
5      person =int(input('请出拳: 【0 石头 1 剪刀2布】\n'))
6      computer=random.randint(0,2)#随机数范围
7      if person==0 and computer==1:
8          print('你真厉害')
9          pass
10     elif person==1 and computer==2:
11         print('你可真棒')
12         pass
13     elif person==2 and computer==0:
14         print('你真厉害')
15         pass
16     elif person==computer:
17         print('不错嘛，竟然平手')
18         pass
19     else:
20         print('你输了')
21         pass
22     count+=1

```

### 九九乘法表

```

1  #打印九九乘法表
2  i=1
3  while i<=9:
4      j=1
5      while j<=i:
6          print('%d*%d=%2d'%(i,j,i*j),end=" ")
7          j+=1
8          pass
9      print()
10     i+=1
11     pass

```

## 直角三角形

```

1  #直角三角形
2  row=9
3  while row>=1:
4      j=1
5      while j<=row:
6          print('*',end=' ')
7          j+=1
8          pass
9      print()
10     row-=1
11     pass

```

## 等腰三角形

```

1  #等腰三角形
2  i=1
3  while i<=10:
4      j1=1
5      while j1<=10-i:#控制打印没空格的数量
6          print(' ',end='')
7          j1+=1
8          pass
9      k=1
10     while k<=2*i-1:#控制打印*号
11         print('*',end='')
12         k+=1
13     print()
14     i+=1
15     pass

```

## 猜年龄小游戏

有三点要求：

- 1.允许用户最多尝试3次
- 2.每尝试3次后，如果还没有猜对，就问用户是否还想继续玩，如果回答Y或y,就让继续猜3次，以此往后，如果回答N或n，就退出程序
- 3.如果猜对了，直接退出

```

1  # # import random

```

```

2  #
3  # # for i in range(3):
4  #     # computer = random.randint(20, 22)
5  #     # person = int(input('请输入年龄: '))
6  #     # if person == computer:
7  #     #     print('恭喜您答对了')
8  #     #     break
9  #     #
10 #     # else:
11 #     #     print('猜错了')
12 #     #     zimu=input('是否继续: 继续请输入Y或者y 结束请输入N或n')
13 #     #     if zimu=='Y' or zimu=='y':
14 #     #         pass
15 #     #     elif zimu=='N' or zimu=='n':
16 #     #         exit()
17 time =0
18 count=3
19 while time<=3:
20     age =int(input('请输入您要猜的年龄: '))
21     if age ==25:
22         print('您猜对了: ')
23         break
24     elif age >25:
25         print('您猜大了')#     else:
26         print('您猜小了')
27     time+=1
28     if time==3:
29         choose=input('还想继续吗? Y/N')
30         if choose == 'Y' or choose=='y':
31             time=0#重置为初始值
32         elif choose=='N' or choose=='n':
33             exit()
34         else:
35             print('请输入正确的符号..谢谢配合')
36

```

小王身高1.75，体重80.5kg,请根据BMI公式，【体重除以身高的平方】，帮助小王计算他的BMI指数，并根据BMI指数

低于18.5 过轻  
 18.5-25 正常  
 25-28 过重  
 28-32 肥胖  
 高于32 严重肥胖  
 用if-elif 判断并打印出来

```

1  high=1.75
2  weight=80.5
3  BMI=weight/(high*high)
4  print('BMI的数据是{}'.format(BMI))
5  if BMI<18.5:
6      print('过轻')
7  elif 18.5<=BMI<25:
8      print('正常')
9  elif 25<=BMI<28:

```

```

10     print('过重')
11 elif 28<=BMI<32:
12     print('肥胖')
13 else:
14     print('严重肥胖')
15 ## Python数据类型
16 >掌握Python的基本数据类型
17 * 数字
18 * int(有符号整数)
19 * long(长整数)(Python3版本取消了)
20 * float(浮点型)
21 * complex(复数)
22 * bool(布尔值)
23     * 返回 True False
24 * 字符串
25 * 字典
26 * 元组
27 * 列表
28 ```python
29 a=[]
30 print(type(a)) #type方法可以查看变量的类型
31 b=10
32 print(type(b))

```

## 03、Python数据类型

- 序列:一组按顺序排列的值，【数据集合】
- 序列优点: 支持索引和切片操作
- 特征:第一个正索引为0，指向的是左端， 第一个索引为负数，指向的是右端
- 切片是指截取字符串中的某一段内容，
- 切片使用语法: [起始下标: 结束下标: 步长] 切片截取的内容不包含结束下标对应的数据，步长指的是隔几个下标获取一个字符
- 下标会越界，切片不会:

### 字符串

#### 不可变的字符序列

- 字符串驻留机制:

仅仅保存一份相同且不可变字符串的方法，不同的值被存放在字符串的驻留池中，Python的驻留机制对相同的字符串只保留一份拷贝，后续创建相同字符串时，不会开辟新空间，而是把该字符串的地址赋给新创建的变量

在需要进行字符串**拼接时建议使用str类型的join方法**，而非+，因为join()方法是先计算出所有字符串中的长度，然后再拷贝，只new一次对象，效率要比+'效率高

```

1 a = 'python'
2 b = "Python"
3 c = '''Python'''
4 print(a, id(a))
5 print(b, id(b))
6 print(c, id(c))
7
8 """结果如下"""
9 python 2090039786608
10 Python 2089748023280
11 Python 2089748023280

```

- 查询操作的方法

```

1 """index() rindex() find() rfind()"""
2 s='hello,hello'
3 print(s.index('lo'))#从头找到的第一次出现的
4 print(s.find('lo'))#从头找到的第一次出现的
5 print(s.rindex('lo'))#从头开始找，找最后一次出现的
6 print(s.rfind('lo'))#从头开始找，找最后一次出现的
7
8 # print(s.index('k'))#index()方法找不到会报错
9 #find()方法找不到会返回-1
10 print(s.find('k'))
11
12
13 """结果如下"""
14 3
15 3
16 9
17 9
18 -1

```

- 大小写转换

```

1 s = 'hello,Python'
2 q = s.upper() # 转成大写后，会产生一个新的字符串对象
3 print(q, id(q))
4 print(s, id(s))
5 print(s.lower(), id(s.lower())) # 转换小写后也会转成一个新的对象
6 print(s, id(s))
7 s2 = 'hello,Python'
8 print(s2.swapcase()) # 所有的大写转小写，所有的小写转大写
9 print(s2.title()) # 把每个单词的第一个字符转成大写，把每个单词剩余的字符串转成小写
10 print(s2.capitalize()) # 第一个字符转大写，剩余的单词 转成小写
11

```

- 字符串内容对齐的操作方法

方法名	作用
center()	居中对齐，第1个参数指定宽度，第2个参数指定填充符，第2个参数是可选的，默认是空格，如果设置宽度小于实际宽度则返回原字符串

方法名	作用
ljust()	左对齐，第1个参数指定宽度，第2个参数指定填充符，第2个参数是可选的，默认是空格，如果设置的宽度小于实际的宽度则返回原字符串
rjust()	右对齐，第1个参数指定宽度，第2个参数指定填充符，第2个参数是可选的，默认是空格，如果设置的宽度小于实际的宽度则返回原字符串
zfill()	右对齐，左边用0填充，该方法返回只接受一个参数，用于指定字符串的宽度，如果指定的宽度小于等于字符串 长度，返回字符串本身

```

1 print(s2.center(20, '*'))
2 """左对齐"""
3 print(s2.ljust(20, '*'))
4 print(s2.ljust(20))
5 """右对齐"""
6 print(s2.rjust(20, '*'))
7 print(s2.rjust(20))
8 print(s2.rjust(8))
9 """右对齐，使用0填充"""
10 print(s2.zfill(20))
11 print(s2.zfill(10))
12 print('-8910'.zfill(8))
13
14 """结果如下"""
15 ****hello,Python****
16 hello,Python*****
17 hello,Python
18 *****hello,Python
19         hello,Python
20 hello,Python
21 0000000hello,Python
22 hello,Python
23 -0008910#用0填充至8位
24

```

### • 字符串分割 操作的方法

**split():**从字符串的左边开始分割，默认的分隔符是空格字符串，返回的值是一个列表

以通过参数step指定分割字符串的分割数

通过参数maxsplit指定分割字符串时的最大分割次数，在经过了最大分割之后，剩余的子串会单独作为一部分

**rsplit():**跟上述有一样，只不过是从小边开始分割

```

1 s='hello world python'
2 lst=s.split()
3 print(lst)
4 s1='hello|world|python'
5
6 print(s1.split(sep='|',maxsplit=1))
7
8 """rsplit()从右侧开始分割"""

```

```

9 print(s.rsplit())
10 print(s1.rsplit('|',maxsplit=1))
11
12 """结果如下"""
13 ['hello', 'world', 'python']
14 ['hello', 'world|python']
15 ['hello', 'world', 'python']
16 ['hello|world', 'python']

```

### • 判断字符串操作方法

方法名	作用
isidentifier()	判断指定的字符串是不是合法的标识符
isspace()	判断指定的字符串是否全部由空白字符组成（回车换行，水平指标符）
isalpha()	判断指定的字符串是否全部由字母组成
isdecimal()	判断指定的字符串是否全部由十进制的数字组成
isnumeric()	判断指定的字符串是否全部由数字组成
isalnum()	判断指定的字符串是否全部由字母和数字组成
isdigit()	判断是否是数字

print(str.startswith('l'))#返回布尔型 判断是否以什么什么开始

print(str.endswith('i'))#判断是否以什么什么结尾

```

1 s='hello,pytohn'
2 print('1',s.isidentifier())
3 print('2','hello'.isidentifier())
4 print('3','\t'.isalpha())
5 print('4','\t'.isspace())
6 print('5','python'.isalpha())
7 print('6','132'.isdecimal())
8 print('7','132死'.isdecimal())
9 print('8','456'.isnumeric())
10 print('9','abc1'.isalnum())
11 print('10','张三ada1'.isalnum())
12
13 """结果如下"""
14 1 False
15 2 True
16 3 False
17 4 True
18 5 True
19 6 True
20 7 False
21 8 True
22 9 True
23 10 True

```

### • 字符串替换



方法名	作用
replace()	第一个参数指定被替换的子串，第2个参数指定替换字串的字符串，该方法返回替换后得到的字符串，，替换前的字符串不发生变化，调用该方法时可以通过第3个参数指定最大替换次数

```

1 s='hello,python'
2 print(s.replace('python','Java'))
3 s2='hello python python python'
4 print(s2.replace('python','Java',2))
5
6
7 """结果如下"""
8 hello,Java
9 hello Java Java python
10

```

## • 字符串比较操作

```

1 """
2 运算符: > >= < <= == !=
3 比较规则: 首先比较两个字符串中的第一个字符，如果相等则继续往下比较，直到两个字符串中的字符串
  不相等时，其比较结果就是两个字符串的比较结果，两个字符串中的所有后续字符将不再继续比较
4
5 比较原理:
6     两个字符进行比较，比较的是(ordinal value)原始值，调用内置函数ord可以得到指定字符的
  ordinal value。与内置函数ord对应的内置函数是chr，调用内置函数chr时指定ordinal value可
  以得到其对应的字符
7 """

```

```

1 print('apple' > 'app')
2 print('apple' > 'banana')#False 相当于 97>98 返回False
3 print(ord('a'),ord('b'))
4
5
6 print(ord('张'))
7 print(chr(24352))
8 print(chr(97),chr(98))
9 """
10 == 与 is 的区别
11     ==比较的是 value
12     is比较的是 di是否相等
13 """
14 a=b='python'
15 c='python'
16 print(a==b)
17 print(b==c)
18 print(a is b)
19 print(c is b)
20 print(id(a))
21 print(id(b))
22 print(id(c))
23

```

```

24 """结果如下"""
25 True
26 False
27 97 98
28 24352
29 张
30 a b
31 True
32 True
33 True
34 True
35 2163358525104
36 2163358525104
37 2163358525104

```

## • 字符串合并

方法名	作用
join()	将列表或元组中的字符串合并成一个字符串

```

1 s=['hello','pytohn','java']
2 print('|'.join(s))
3 t=('hello','world')
4 print('*'.join(t))
5 print('*'.join('python'))
6
7 """结果如下"""
8 hello|pytohn|java
9 hello*world
10 p*y*t*h*o*n

```

## • 格式化字符串3中方法

### (1) %作占位符

```

1 name='李四'
2 age=18
3 print('%s的今年%s岁'%(name,age))

```

### (2) {}作占位符

```

1 name='李四'
2 age=18
3 print('{}的今年{}岁'.format(name,age))

```

### (3) f-string

```

1 name='李四'
2 age=18
3 print(f'{name}的今年{age}岁')

```

## 宽度精度的设置

```
1 print("%d"%99)
2 print('%10d'%100)#10表示宽度
3 print('%.3f'%3.1415926)#.3表示小数点后三位
4 print("%10.3f"%3.1415926)#总宽度为10，小数点后有3位
5 print('{0:.3}'.format(3.1415926))#.3表示一共3位
6 print('{:.3f}'.format(3.1415926))#.3f表示3位小数
7 print('{:10.3f}'.format(3.14152949))#同时设置宽度和精度，一共10位，小数点后有3位
8
9 """结果如下"""
10 99
11      100
12 3.142
13      3.142
14 3.14
15 3.142
16      3.142
17
```

### • 去除空格

```
1 a='      python      '
2 print(a.strip())#去除两边的空格
3 print(a.lstrip())#去除左边空格
4 print(a.rstrip())#去除右边空格
5 #赋值字符串
6 b=a
7 print(b)
8 print(id(a))#查看内存地址
9 print(id(b))
```

### • 索引切片

```
1 st='hello world'
2 print(st[0])
3 print(str[::-1])#逆序输出
4 print(st[2:5])#从下标2开始不包含5【左闭右开】
5 print(st[2:])#从第三个字符到最后
6 print(st[0:3])#从0开始可以省略0
7 print(st[:3])
```

### • 编码与解码

```
1 s = '天涯共此时'
2 #编码
3 print(s.encode(encoding='GBK')) # GBK编码格式中，一个中文占两个字节
4 print(s.encode(encoding='UTF-8')) # 在UTF-8编码格式中，一个中文占三个字节
5 #解码
6 byte=s.encode(encoding='GBK')
7 print(byte.decode('GBK'))
```

## 列表

- 特点:

"""

支持增删改查

列表中的数据是可以变化的，【数据项可以变化，内存地址不会改变】

用[]来表示列表类型，数据项之间用逗号来分割，注意：数据项可以说任意的数据类型

"""

```
1 list=[]#空列表
2 print(list)
3 print(len(list))#获取列表对象 中数据个数
4 str='lodsas'
5 print(len(str))
```

- 查找

```
1 listA=['abcd',123,456,798,True,12.255]
2 print(listA)
3 print(listA[0])#找出第一个元素
4 print(listA[1:3])
5 print(listA[3:])
6 print(listA[::-1])#倒叙
7 print(listA*3)#输出多次列表中数据【复制多次】
```

- 添加

```
1 print('-----增加-----')
2 listA.append('Jay')
3 listA.append([1,1,2,'dfgs',{'a1':12}])
4 print(listA)
5 listA.insert(1,'插入的元素')#插入操作，需要指定插入位置
6 # reData=list(range(10))#强制转换list对象
7 listA.extend([12,45,484])#扩展，批量添加
8 print(listA)
```

- 修改

```
1 print('-----修改-----')
2 print('修改之前',listA)
3 listA[0]='Peter'
4 print('修改之后',listA)
```

- 删除

```

1 print('-----删除-----')
2 del listA[0] #删除列表中第一个元素
3 print(listA)
4 del listA[1:3] #批量删除
5 print(listA)
6 listA.remove(798) #指出指定元素
7 print(listA)
8 listA.pop(0) #移除第一个元素 #根据下标来移除
9 print(listA)
10

```

- 列表生成式

```

1 lst=[i*i for i in range(6)]
2 print(lst)

```

## 元组

- 元组是**不可变序列**，在创建后不能做任何修改，没有增删改操作
- 用()来创建元组类型，数据项用逗号来分割
- 可以是任何类型
- 当元组中**只有一个元素时，要加上逗号**，不然解释器会当作本身的类型处理
- 同样支持切片操作

```

1 t=(10,[10,30,40],9)
2 print(t)
3 print(type(t))
4 print(t[0],type(t[0]),id(t[0]))
5 print(t[1],type(t[1]),id(t[1]))
6 print(t[2],type(t[2]),id(t[2]))
7 #t[1]=100报错，元祖是不允许修改元素的
8 """由于[10,30,40]是列表，而列表是可变序列，所以可以向列表中添加元素，而列表的内存地址不变"""
9 t[1].append(100) #向列表中添加元素
10 print(t,id(t[1]))
11
12 """结果如下"""
13 #(10, [10, 30, 40], 9)
14 #<class 'tuple'>
15 #10 <class 'int'> 2775649288720
16 #[10, 30, 40] <class 'list'> 2775650645440
17 #9 <class 'int'> 2775649288688
18 #(10, [10, 30, 40, 100], 9) 2775650645440
19

```

- 创建

```

1  """第一种方式：使用()"""
2  t=('pytohn','hello',454)
3  print(type(t))
4  """第二种方式创建使用内置函数tuple()"""
5  t=tuple(('hello',45,'world'))

```

#### • 查找

```

1  tupleA=()#空列表
2  tupleA=('abcd',123,12.55,[123,45,'45hf'])
3  print(tupleA)
4  print(tupleA[2])
5  print(tupleA[1:3])
6  print(tupleA[::-1])
7  print(tupleA[::-2])
8  print(tupleA[-2:-1])#倒着取下标， -2到-1区间的数据
9  print(tupleA[-4:-1])

```

#### • 对元组中的列表元素进行修改

```

1  tupleA[3][0]=250#对元组中的列表类型的数据进行修改
2
3  tupleB=tuple(range(10))
4  print(tupleB.count(9))

```

#### • 遍历

```

1  t=('python',20,[10,20,2])
2  for item in t:
3      print(item)

```

## 字典

- 字典不是序列类型，没有下标的概念，是一个**无序**的键值集合，是内置的高级数据类型
- 不可变序列
- 字典可以存储任意对象
- 字典以键值对的形式创建的{key:value}利用大括号包裹
- 字典中某个元素时，是根据键，值 字典的每个元素由2部分组成：键：值
- 访问值的安全方法**get()方法**，在我们不确定字典中是否存在某个键而又想获取其值时，可以使用get()方法，还可以设置默认的值
- 字典创建**

```

1  """最常用方式：使用花括号"""
2  socres={'张三':100,'李四':520,'王五':51}
3  """使用内置函数dict()"""
4  dict(name='jack',age=20)

```

#### • 获取元素

如果查找的键不存在，使用dictA[]的方式会报错，

但是使用.get()的方式不会报错，会返回一个None

```
1 dictA={"pro":"艺术专业","school":"'上海戏剧学院'"}
2 print(dictA['pro'])
3 print(dictA.get('pro'))
```

- 键的判断

存在返回True

不存在返回False

```
1 dictA={"pro":"艺术专业","school":"'上海戏剧学院'"}
2 print('pro' in dictA)
3 print('pro' not in dictA)
```

- 添加，修改，获取字典数据

```
1 dictA={"pro":"艺术专业","school":"'上海戏剧学院'"}
2 dictA['name']='李易峰'
3 dictA['age']=29
4 print(dictA)
5 print(len(dictA))
6 print(dictA['name'])#通过键获取值
7 dictA['name']='成龙'#修改键对应的值
8 print(dictA)
9 print(dictA.keys())#获取所有的键
10 print(dictA.values())#获取所有的值
11 print(dictA.items())#获取所有的键和值
12 for key,value in dictA.items():
13     # print(item)
14     print('%s值==%s'%(key,value))
15
16 dictA.update({'age':32})#更新 存在就更新，不存在就添加
17 dictA.update({"height":1.78})#添加
18 print(dictA)
19
```

- 删除

```
1 del dictA['name']
2 dictA.pop('age')
3 print(dictA)
4 dictA.clear()#清空字典元素
```

- 内置函数zip():

- 用于将可迭代的对象作为参数，将对象中对应的元素打包成一个元组，然后返回由这些元组组成的列表

```
1 items = ['Fruit', 'Books', 'Others']
2 prices = [96, 88, 65]
3 lst=zip(items,prices)
4 print(list(lst))
```

- 字典生成式

```
1 items = ['Fruit', 'Books', 'Others']
2 prices = [96, 88, 65]
3 d = {item.upper(): price for item, price in zip(items, prices)}
4 print(d)
5
```

---

## 共有方法

```
1 #+ 合并
2 strA='人生苦短'
3 strB='我用Python'
4 print(strA+strB)
5 List_A=list(range(10))
6 list_B=list(range(10,21))
7 print(List_A+list_B)
8 # * 复制
9 print(strA*3,end=' ')
10 print(List_A*3)
11
12 # in 判断对象是否存在
13 print('我' in strA)
14 print(10 in list_B)
```

## 不可变序列与可变序列

不可变序列：字符串、元组

- 不可变序列：没有增、删、改操作，

```
1 """不可变：元组，字符串"""
2 s='hello'
3 print(id(s))#2213691636720
4 s=s+'world'
5 print(id(s))#2213691680432
6
7 """对象地址发生改变所以是不可变序列"""
```

可变序列：列表、字典



- 可变序列：可以对序列执行增、删、改操作，**对象地址不发生改变**

```

1  """可变：列表，字典"""
2  lst=[10,20]
3  print(id(lst))#2250871125440
4  lst.append(100)
5  print(id(lst))#2250871125440
6  """
7  添加元素后对象地址不变，所以是可变序列
8  """

```

## set集合

- set 不支持索引切片，是一个**无序**的且**不重复**的容器
- 类似于字典，但是只有Key,没有value
- 与列表、字典一样都属于可变类型的序列
- 创建

```

1  """第一种方式使用{}"""
2  t={20,20,202,30,202,70}
3  print(t)#{202, 20, 70, 30},集合中的元素不允许重复
4
5  """第二种方式set()"""
6  s=set(range(6))
7  print(s)
8  s2=set([1,2,3,4,5,6,8,7])
9  print(s2)
10 s3=set((1,2,3,4,5,6,8,7))
11 print(s3)
12 s4=set('pytohn')
13 print(s4)

```

- 判断操作

```

1  """in      not in """
2  s={10,20,111,220,12020,1515}
3  print(10 in s)
4  print(7 not in s)

```

- 添加操作

方法名	作用
add()	一次添加一个元素
update()	一次至少添加一个元素

```

1  """add()  update()"""
2  s={10,20,11,22,12}
3  s.add(80)#一次添加一个元素
4  print(s)
5  s.update({500,800,8080})#一次至少添加一个元素
6  print(s)
7  s.update([100,200,5056])
8  s.update((78,99,46))
9  print(s)

```

- 删除操作

方法名	作用
remove()	一次删除一个指定元素，如果元素不存在，抛出异常KeyError
discard()	一次删除一个指定元素，如果元素不存在，不抛出异常
pop()	一次只删除一个任意元素
clear()	清空集合

```

1  """remove()  discard()  pop()  clear()"""
2  s.remove(100000)#KeyError
3  print(s)
4  s.discard(1)
5  s.pop()#随机删除元素,不能指定元素
6  s.clear()

```

- 两个集合是否相等

```

1  s={10,20,30,40,50}
2  s2={10,30,40,20,50}
3  print(s1 == s2)#True
4  print(s1 != s2)#False

```

- 一个集合是否是另一个集合的子集

```

1  s1={10,20,30,40,50,60}
2  s2={10,20,30,40}
3  s3={10,20,80}
4  print(s2.issubset(s1))#True
5  print(s3.issubset(s1))#False

```

- 一个集合是否是另一个集合的超集

```

1  print(s1.issuperset(s2))#True
2  print(s1.issuperset(s3))#False

```

- 两个集合是否含有交集

```
1 print(s2.isdisjoint(s3))#False
2 有交集为False
```

## • 集合的数学操作

### ◦ 交集

```
1 s1={10,20,30,40}
2 s2={20,30,50,40,60}
3 print(s1.intersection(s2))
4 print(s1 & s2)
```

### ◦ 并集

```
1 print(s1.union(s2))
2 print(s1 | s2)
```

### ◦ 差集

```
1 print(s1.difference(s2))#a中有的b中没有的
2 print(s1 - s2)
```

### ◦ 对称差集

```
1 print(s1.symmetric_difference(s2))
2 print(s1 ^ s2)
```

## • 集合生成式

```
1 """将{}修改为[]就是列表生成式"""
```

没有元组生成式

```
1 s={i*i for i in range(6)}
2 print(s)
```

## 总结

数据结构	是否可变	是否重复	是否有序	定义符号
列表(list)	可变	可重复	有序	[]
元组(tuple)	不可变	可重复	有序	()
字典(dict)	可变	key不可重复, value 可重复	无序	{key:value}
集合(set)	可变	不可重复	无序	{}
字符串(str)	不可变	可重复	有序	"/"

---

# 04、Python函数

---

## 4.1、函数初识

- 在编写程序的过程中，有某一功能代码块出现多次，但是为了提高编写的效率以及代码的重用，所以把具有独立功能的代码块组织为一个小模块，这就是函数
- 就是一系列Python语句的组合，可以在程序中运行一次或者多次，
- 一般具有独立的功能
- 代码的复用最大化以及最小化冗余代码，整体代码结构清晰，问题局部化
- 语法结构

```
1 def 函数名(参数1, 参数2, 参数3):  
2     """文档说明"""  
3     函数体  
4     return 值
```

- 函数定义

```
1 def pritInfo():  
2     '''  
3     这个函数用来打印小张信息  
4     '''  
5     #函数代码块  
6     print('小张的身高是%f'%1.73)  
7     print('小张的体重是%f'%130)  
8     print('小张的爱好是%s'% '周杰伦')  
9     print('小张的专业是%s'% '信息安全')
```

定义函数发生的事情：

1. 申请内存空间保存函数体代码
2. 将上述代码地址绑定给函数名
3. 定义函数不会执行函数体代码，但是会检测函数体语法

- 函数 调用

```
1 print(pritInfo())#打印的是内存地址  
2 pritInfo()
```

调用函数发生的事情：

1. 通过函数名找到函数的内存地址
2. 然后加括号就是在触发函数体代码的执行

```
1 def foo():  
2     bar()  
3     print('from foo')  
4 def bar():
```

```

5     print('from bar')
6
7     foo()
8     """执行过程"""
9     函数的使用分两个 阶段，
10    第一个阶段：定义
11    第二个阶段：调用
12    定义的时候只检测语法，不执行代码
13    在定义foo时咩有语法错误
14    然后再去定义bar，
15    最后再去调用foo()时，bar早就丢到内存中了，所以就不会报错，
16

```

```

1     def foo():
2         bar()
3         print('from foo')
4
5     foo()
6
7     def bar():
8         print('from bar')
9
10    """会报错"""
11

```

- 进一步去输出不同人的信息， 通过传入参数来解决

```

1     def pritInfo(name,height,weight,hobby,pro):
2
3         #函数代码块
4         print('%s的身高是%f'%(name,height))
5         print('%s的体重是%f'%(name,weight))
6         print('%s的爱好是%s'%(name,hobby))
7         print('%s的专业是%s'%(name,pro))

```

- 调用带参数的信息

```

1     pritInfo('peter',175,130,'听音乐','甜品师')
2     pritInfo('小李',189,140,'打游戏','软件工程师')

```

## 4.2、函数参数

- 参数分类：

必选参数，默认参数【缺省参数】，可选参数，关键字参数

- 参数：其实就是函数为了实现某项特定功能，进而为了得到实现功能所需要的数据

```

1
2 def sum(a,b):#形式参数：只是意义上的一种参数，在定义的时候是不占内存地址的
3
4 sum=a+b
5 print(sum)

```

- **必选参数**（位置参数）

函数调用的时候，必选参数是必须要赋值的

sum(20,15)#20,15就是实际参数 实参， 实实在在的参数，是实际占用内存地址的  
sum()不能这样写，需要传参

位置实参：在函数调用阶段，按照从左往右的顺序依次传入值

**位置实参必须放在关键字实参前，否则报语法错误**

在调用阶段，实参（变量值）会绑定给形参（变量名）

实参与形参的绑定关系：在函数调用时生效，调用结束后解除绑定关系

- **默认参数【缺省参数】**（默认形参）

在定义函数阶段，就已经被赋值的形参，称之为默认形参

**特点：在定义阶段就已经被赋值**，意味着在调用阶段可以不用为其赋值

位置形参必须在默认形参的左边

```

1 def sum(a=1,b=11):
2     print(a+b)
3
4 sum()#在调用的时候，如果未赋值，就用定义函数时给定的默认值
5 sum(10)#10赋给了a,b还是11,结果为21

```

- **关键字参数**

关键字实参：在函数调用阶段，按照key=value的形式传入值

**位置实参必须放在关键字实参前，否则报语法错误**

指名道姓的给某个形参传值，

```

1 def a(a,b):
2     res=a+b
3     return res
4 print(a(b=2,a=1))

```

- **可变参数**（当参数的个数不确定时，比较灵活）

**\*形参名**：用来接受溢出的位置实参，溢出的位置实参会被 \* 保存成**元组形式**

```

1 def getComputer(a,b*args):
2     '''
3     计算累加和

```

```

4     '''
5     # print(args)
6     result=0
7     for item in args:
8         result+=item
9     print(result)
10
11 getComputer(1,2,3,4)
12
13 """结果如下"""
14 1 2 (3,4)

```

- \*可用在实参中，实参中带\*，把\*后的值打散成位置实参

```

1 def func(x,u,z):
2     print(x,u,z)
3 func(*[11,22,33])#相当于func(11,22,33)
4
5 def func2(x,y,*args):
6     print(x,y,args)
7 func2(*'hello')
8 """结果如下"""
9 11 22 33
10 h e ('l','l','o')

```

- 形参实参都带\*

```

1 def func(x,y,*args):#args=(3,4,5,6)
2     print(x,y,args)
3 func(1,2*[3,4,5,6])#相当于func(1,2,3,4,5,6)
4
5 """结果如下"""
6 1 2 (3,4,5,6)

```

- 关键字可变参数

1. \*\*来定义
2. 在函数体内 参数关键字是一个字典类型，key是一个字符串
3. 可变参数指的是在调用函数的时候，传入的值（实参）的个数不固定

```

1 def Func(a,b,**kwargs):
2     print(a,b,kwargs)
3 Func(1,b=2,x=3,b=5)
4
5 def Infofunc(*args,**kwargs): #可选参数必须在关键字可选参数之前
6     print(args)
7     print(kwargs)
8 Infofunc(1,2)
9 # Infofunc(12,13,46,age=18)

```

- \*\*可用在实参中，实参中带\*\*，把\*\*后的值打散成关键字实参

```

1 def func(x,y,c):
2     print(x,y,c)
3     #func(*{'x':1,'y':2,'c':3})#func('x','y','c')
4     func(**{'x':1,'y':2,'c':3})#func(x=1,y=2,c=3)
5
6
7     """结果如下"""
8     1 2 3

```

- 形参与实参都带\*\*

```

1 def func(x,y,**kwargs):
2     print(x,y,kwargs)
3     func(**{'y':111,'x':211,'z':456,'d':789})#打散后相当于
4     func(y=222,x=211,z=456,d=789)
5
6     """结果如下"""
7     211 111 {'z':456,'d':789}

```

- 混用\*与\*\*: \*args 必须在 \*\*kwargs 前面

```

1 def fun(*args,**kwargs):
2     print(args)
3     print(kwargs)
4     fun(1,2,3,4,5,6,8,x=1,y=2,z=3)
5
6     """结果如下"""
7     (1,2,3,4,5,6,8)
8     {'x':1,'y':2,'z':3}

```

- 命名关键字参数 (了解)

```

1 def func(x,y,*a,b):#a和b为命名关键字参数
2     print(x,y)
3     print(a,b)

```

## 案例实现

- 接收N个数字，求这些参数数字的和



```

1 def num(*args):
2     '''
3     #写函数，接收N 个数字，求这些参数数字的和
4     '''
5     result=0
6     for i in args:
7         result+=i
8
9     return result
10 a=num(1,2,2,2,2)
11 print(a)
12
13

```

- 找出传入的列表或元组的奇数位对应的元素，并返回一个新的列表

```

1 def Func1(con):
2     '''
3     找出传入的列表或元组的奇数位对应的元素，并返回一个新的列表
4     '''
5     listA=[]
6     index=1
7     for i in con:
8         if index%2==1:#判断奇数位
9             listA.append(i)
10            index+=1
11    return listA
12 # e=Func1([1,2,45,1,1,'as',{1,25},'hs'])
13 e=Func1(list(range(1,11)))
14 print(e)

```

- 检查传入字典的每一个value的长度，如果大于2，那么仅仅保留前两个长度的内容，并将新的内容返回给调用者，PS:字典中的数据只能是字符串或者是列表

```

1 def Func2(dic):
2     '''
3     检查传入字典的每一个value的长度，如果大于2，那么仅仅保留前两个长度的内容，
4     并将新的内容返回给调用者，PS:字典中的数据只能是字符串或者是列表
5     '''
6     result={}
7     for key,value in dic.items():
8         if len(value)>2:
9             result[key]=value[:2]#向字典添加新的数据
10        else:
11            result[key]=value
12
13    return result
14 #调用
15 dicobj={'name':'张三','hobby':['唱歌','跳舞','编程'],'major':'信息安全'}
16 print(Func2(dicobj))

```

### 4.3、函数返回值

- 概念：函数执行完以后会返回一个对象，如果在函数的内部有return，就可以返回实际的值，否则返回None
- 类型：可以返回任意 类型，返回值类型应该取决于return 后面的类型
- 用途：给调用方返回数据
- 在一个函数体内可以出现多个return，但是肯定只有一个return
- 如果在一个函数体内，执行了return,意味着就执行完成推出了，return后面的代码将不会执行

```
1  #返回值
2
3  def Sum(a,b):
4      sum=a+b
5      return sum #将计算结果返回
6  result=Sum(12,15)#将返回的值赋给其他变量
7  print(result)
8
9  def calComputer(num):
10     '''
11     累加和
12     '''
13     listA=[]
14     result=0
15     i=1
16     while i<=num:
17         result+=i
18         i+=1
19         # listA.append(result)
20     listA.append(result)
21     return listA
22 message=calComputer(10)
23 print(type(message))
24
25 print(message)
26
27
28 def returnType():
29     '''
30     返回元组类型数据
31     '''
32     return 1,2,3
33 a =returnType()
34 print(type(a))
35 print(a)
36
37 def returnType1():
38     '''
39     返回字典类型
40     '''
41     return {"name":"成龙"}
42 b=returnType1()
43 print(type(b))
44 print(b)
```

## 4.4、名称空间与作用域

### (1) 名称空间 (namespaces)

用来存放名字的地方是对栈区的划分

有了名称空间，就可在栈区中放相同的名字

- 1.1、内置名称空间

**存放的名字**：是Python解释器内置的名字

```
1  """
2  交互模式下输入：
3  >>> input
4  <built-in function input>
5  >>> print
6  <built-in function print>
7  >>>
8
9  """
```

**存活周期**：Python解释器启动则产生，Python解释器关闭则销毁

- 1.2、全局名称空间

**存放的名字**：只要不是函数内定义的，也不是内置的，剩下的都是全局名称空间

**存活周期**：Python文件执行则产生，Python文件运行完毕则销毁

- 1.3、局部名称空间

**存放的名字**：在调用函数时，运行函数体代码过程中产生的函数内的名字

**存活周期**：在函数调用时存活，函数调用完毕后则销毁

- 1.4、名称空间加载顺序

```
1  """
2  内置名称空间》全局名称空间》局部名称空间
3  """
```

- 1.5、销毁顺序

```
1  """
2  局部名称空间》全局名称空间》内置名称空间
3  """
```

- 1.6、名字查找优先级

```

1  """当前所在的位置向上一层一层查找"""
2
3  """
4  如果当前在局部名称空间：
5      局部名称空间》全局名称空间》内置名称空间
6
7  如果当前在全局名称空间：
8      全局名称空间》内置名称空间
9  """

```

#### 示范1:

```

1  def func():
2      print(x)
3  x=111
4  func()
5
6  """结果如下"""
7  111

```

**示范2:** 名称空间的“嵌套”关系是**以函数定义阶段为准**，与调用位置无关

```

1  x=1
2  def func():
3      print(x)
4
5  def foo():
6      x=222
7      func()
8  foo()
9
10 """结果如下"""
11 1

```

#### 示范3: 函数嵌套定义

```

1  input=111
2  def f1():
3      input=222
4      def f2():
5          input=333
6          print(input)
7      f2()
8  f1()

```

#### 示范4:

```

1 x=111
2 def fun():
3     print(x)
4     x=222
5 fun()
6 """结果报错"""

```

## (2) 作用域

### 作用范围

#### • 2.1、全局作用域

全局作用域：内置名称空间。全局名称空间

```

1 """
2 全局存活
3 全局有效：被所有函数共享
4 """

```

#### • 2.2、局部作用域

局部作用域：局部名称空间的名字

```

1 """
2 临时存活
3 局部有效
4 """

```

#### • 2.3、作用域与名字查找的优先级

在局部作用域查找名字时，起始位置是局部作用域，所以先查找局部名称空间，没有找到，再去全局作用域查找：先查找全局名称空间，没有找到，再查找内置名称空间，最后都没有找到就会抛出异常

```

1 x=100 #全局作用域的名字x
2 def foo():
3     x=300 #局部作用域的名字x
4     print(x) #在局部找x
5 foo() #结果为300

```

在全局作用域查找名字时，起始位置便是全局作用域，所以先查找全局名称空间，没有找到，再查找内置名称空间，最后都没有找到就会抛出异常

```

1 x=100
2 def foo():
3     x=300 #在函数调用时产生局部作用域的名字x
4     foo()
5     print(x) #在全局找x, 结果为100

```

在函数内，无论嵌套多少层，都可以查看到全局作用域的名字，若要在函数内修改全局名称空间中名字的值，当值为不可变类型时，则需要用到`global`关键字

```

1 x=1
2 def foo():
3     global x #声明x为全局名称空间的名字
4     x=2
5 foo()
6 print(x) #结果为2

```

当实参的值为可变类型时，函数体内对该值的修改将直接反应到原值

```

1 num_list=[1,2,3]
2 def foo(nums):
3     nums.append(5)
4
5 foo(num_list)
6 print(num_list)
7 #结果为
8 [1, 2, 3, 5]

```

对于嵌套多层的函数，使用`nonlocal`关键字可以将名字声明为来自外部嵌套函数定义的作用域（非全局）

```

1 def f1():
2     x=2
3     def f2():
4         nonlocal x
5         x=3
6         f2() #调用f2(),修改f1作用域中名字x的值
7         print(x) #在f1作用域查看x
8
9 f1()
10
11 #结果
12 3

```

`nonlocal x`会从当前函数的外层函数开始一层层去查找名字`x`，若是一直到最外层函数都找不到，则会抛出异常。

## 4.5、函数嵌套

```

1 #函数分类：根据函数返回值和函数参数决定
2 #有参数无返回值
3 #有参数有返回值
4 #无参数无返回值
5 #无参数有返回值

```

- 函数的嵌套调用

```

1 def max2(x,y):
2     if x>y:
3         return x
4     else:
5         return y
6 def max4(a,b,c,d):

```

```

7      #第一步: 比较a,b, 得到res1
8      res1=max2(a,b)
9      #第二步: 比较res1,c得到res2
10     res2=max2(res1,c)
11     #第三步: 比较res2,d得到res3
12     res2=max2(res2,d)
13     return res2
14 res=max4(1,2,3,4)
15 print(res)

```

- 函数的嵌套定义

```

1 def f1():
2     def f2():
3         pass

```

## 4.6、函数全局变量

- 局部变量: 就是在函数内部定义的变量【作用域仅仅局限在函数的内部】
- 不同的函数, 可以定义相同的局部变量, 但是各自用各自的 不会产生影响.
- 局部变量的作用: 为了临时的保存数据, 需要在函数中定义来进行存储.
- 当全局变量和局部变量出现重复定义的时候, 程序会优先执行函数定义的内部变量.
- 如果函数内部要想对全局变量进行修改, 必须使用global关键字进行声明.

```

1 # pro='信息安全'#全局变量
2 # name='周润发'
3 # def printInfo():
4 #     name='刘德华'#局部变量
5 #     print('{}'.format(name))
6 #
7 # def TestMethod():
8 #     name='peter'#局部变量
9 #     print(name)
10 # printInfo()
11 # TestMethod()
12 #
13 #
14 # def message():
15 #     print(pro)
16 #
17 # message()
18 #
19 pro='123'
20 def changGlobal():
21     global pro
22     pro='456'
23     print(pro)
24 changGlobal()

```

## 4.7、引用

- Python中所有值的传递，传递都不是值本身，而是值的引用，即内存地址
- 在Python中，值是靠引用来传递来的，可以用id()查看一个对象的引用是否相同.
- id是值保存在内存中那块内存地址的标识.
- **不可变类型**

```
1 a=1
2 def func(x):
3     print('x的地址{}'.format(id(x)))
4     x=12
5     print('修改之后的x 的地址:{}'.format(id(x)))
6
7 print('a的地址: {}'.format(id(a)))
8 func(a)
9
```

- **可变类型**

```
1 li=[]
2 def testRenc(parms):
3     print(id(parms))
4     li.append([1,2,3,54])
5 print(id(li))
6 testRenc(li)
7 print('外部变量对象{}'.format(li))
8
```

- 小结

1. python中，万物皆对象，在函数调用的时候，实参传递的就是对象的引用，
2. 了解了原理之后，就可以更好的去把控，在函数内部的处理是否会影响到外部数据的变化
3. 参数传递是通过对象引用来完成 参数传递是通过对象引用来完成 参数传递是通过对象引用来完成

---

## 4.8、函数对象和闭包

### 函数对象

可以把函数当成变量去用

func=内存地址

```
1 def func():
2     print('from func')
3
4 #1.可以赋值
5 f=func
6 print(f, func)
7
8 #2.可以当作参数传入
9 def foo(x):
10     print(x)
11 foo(func)#foo(func的内存地址)
12
```



```

13 #3.把函数当作另外一个函数的返回值
14 def foo2(x):#x=func的内存地址
15     return x#return func的内存地址
16 res=foo2(func)#foo2(func的内存地址)
17 print(res)
18
19 #4.当作容器类型的一个元素
20 #l=[func,]
21 #l[0]
22
23 dic={'k1':func}
24 print(dic)
25 dic['k1']()
26
27
28 """结果如下"""
29 <function func at 0x0000018533F7FEB0> <function func at 0x0000018533F7FEB0>
30 <function func at 0x0000018533F7FEB0>
31 <function func at 0x0000018533F7FEB0>
32 {'k1': <function func at 0x0000018533F7FEB0>}
33 from func

```

#### 案例:

```

1 def login():
2     print('登录')
3
4
5 def transfer():
6     print('转账')
7
8
9 def check_balance():
10    print('查询余额')
11
12
13 def withdraw():
14    print('提现')
15
16 def register():
17    print('注册')
18 func_dic = {
19
20    '1': login,
21    '2': transfer,
22    '3': check_balance,
23    '4': withdraw,
24    '5':register
25 }
26 # func_dic['1']()
27 while True:
28     print("""
29     0 退出
30     1 登录
31     2 转账

```

```

32     3 查询余额
33     4 提现
34     5 注册
35     """
36     choice = input('请输入编号: ').strip()
37     if not choice.isdigit():
38         print('请输入正确的编号')
39         continue
40     if choice == '0':
41         break
42
43     if choice in func_dic:
44         func_dic[choice]()
45     else:
46         print('输入的编号不正确')
47
48     # if choice == '1':
49     #     login()
50     # elif choice == '2':
51     #     transfer()
52     # elif choice == '3':
53     #     check_balance()
54     # elif choice == '4':
55     #     withdraw()
56     # else:
57     #     print('输入的编号不正确')
58

```

### 案例改进:

```

1  def login():
2      print('登录')
3
4
5  def transfer():
6      print('转账')
7
8
9  def check_balance():
10     print('查询余额')
11
12
13  def withdraw():
14     print('提现')
15
16
17  def register():
18     print('注册')
19
20
21  func_dic = {
22      '0': ['退出', None],
23      '1': ['登录', login],
24      '2': ['转账', transfer],
25      '3': ['查询余额', check_balance],

```

```

26     '4': ['提现', withdraw],
27     '5': ['注册', register]
28 }
29 # func_dic['1']()
30 while True:
31     for i in func_dic:
32         print(i, func_dic[i][0])
33     choice = input('请输入编号: ').strip()
34     if not choice.isdigit():
35         print('请输入正确的编号')
36         continue
37     if choice == '0':
38         break
39
40     if choice in func_dic:
41         func_dic[choice][1]()
42     else:
43         print('输入的编号不正确')
44
45     # if choice == '1':
46     #     login()
47     # elif choice == '2':
48     #     transfer()
49     # elif choice == '3':
50     #     check_balance()
51     # elif choice == '4':
52     #     withdraw()
53     # else:
54     #     print('输入的编号不正确')
55

```

## 闭包函数

1. 嵌套函数
2. 内部函数引用了外部函数的变量
3. 返回值是内部函数

闭包函数=名称空间与作用域+函数嵌套+函数对象

**核心点：**名字的查找关系是以函数定义阶段为准

- "闭"函数指的是该函数的内嵌函数

```

1 def f1():
2     def f2():
3         pass

```

- "包"函数指的是该函数对外层函数作用域名字的引用（不是对全局作用域）

```

1 """闭包函数: 名称空间与作用域+函数嵌套"""

```

```

2  def f1():
3      x=1
4      def f2():
5          print(x)
6          f2()
7  x=1111
8  def bar():
9      x=4444
10     f1()
11  def foo():
12      x=22222
13      bar()
14  foo()

```

```

1  """闭包函数：函数对象"""
2  def f1():
3      x = 33333333
4
5      def f2():
6          print(x)
7
8      return f2#相当于把 f2丢到全局，
9
10
11  f = f1()#f的内存地址就是f2的内存地址
12  print(f)
13
14
15  def foo():
16      x = 5555
17      f()
18
19
20  foo()
21
22
23  """结果如下"""
24  <function f1.<locals>.f2 at 0x0000013F2D2EA950>
25  33333333

```

## 4.9、装饰器

### (1) 什么是装饰器：

- 器指的是工具，可以定义成函数
- 装饰指的是为其他事务添加额外的东西来点缀

上面两者合到一起：

- 装饰器指的是定义一个函数，该函数用来为其他函数添加额外的功能

函数装饰器分为：

- 无参装饰器和有参装饰两种，二者的实现原理一样，都是‘函数嵌套+闭包+函数对象’的组合使用的产物。

## (2) 为何要用装饰器

开放封闭原则

开放：指的是对扩展功能是开放的

封闭：指的是对修改源代码是封闭的

**装饰器就是在不修改被装饰器对象的源代码以及调用方式的前提下，为被装饰对象添加新功能**

## (3) 装饰器实现思路

### 无参装饰器

- 方案一：失败

没有修改被装饰对象的调用方式，但是改变了源代码

```
1 def index(x, y):
2     start = time.time()
3     time.sleep(2)
4     print(' index %s %s ' % (x, y))
5     end = time.time()
6     print(end - start)
7
8 index(1, 2)
9 # index()
```

- 方案二

```
1 # 问题：没有修改被装饰对象的源代码，也没有修改调用方式，但是代码冗余
2 def index(x, y):
3     time.sleep(2)
4     print(' index %s %s ' % (x, y))
5
6
7 start = time.time()
8 index(11, 22)
9 end = time.time()
10
11 start = time.time()
12 index(11, 22)
13 end = time.time()
```

- 方案三

问题:解决了代码冗余,但是函数的调用方式改变了

```
1 def index(x, y):
2     time.sleep(2)
3     print(' index %s %s ' % (x, y))
4
5
6 def wrapper():
```

```

7     start = time.time()
8     index(11, 22)
9     end = time.time()
10
11
12 wrapper()
13 wrapper()
14 wrapper()

```

- 方案三的优化一:

在方案三基础上优化代码: 将Index写活了(参数写活了)

```

1  def index(x, y, z):
2      time.sleep(2)
3      print(' index %s %s %s ' % (x, y, z))
4
5
6
7  def wrapper(*args, **kwargs):
8      start=time.time()
9      index(*args, **kwargs)
10     stop=time.time()
11     print(stop-start)
12 wrapper(11, 22, 44)
13 wrapper(111, y=111, z=4545)

```

- 方案三的优化二:
- 在优化一的基础上把被装饰对象写活,原来只能装饰Index

```

1  def index(x, y, z):
2      time.sleep(2)
3      print(' index %s %s %s ' % (x, y, z))
4
5
6  def home(name):
7      time.time()
8      print('welcome %s to home page' % name)
9
10
11 def outer(func): # func=index的内存地址
12     # func = index
13     def wrapper(*args, **kwargs):
14         start = time.time()
15         func(*args, **kwargs) # index的内存地址()
16         stop = time.time()
17         print(stop - start)
18
19     return wrapper
20
21
22 home=outer(home)
23 home('zhao')
24
25 # f = outer(index) # f=outer(index的内存地址)

```

```

26 # f(x=1, y=2, z=3)
27 index = outer(index) # 偷梁换柱.>>>此时的index指向的是wrapper的内存地址
28 print(index)
29 index(x=1, y=2, z=3)

```

- 方案三的优化三

将wrapper做的跟被装饰器一模一样,以假乱真

```

1  def index(x, y, z):
2      time.sleep(2)
3      print(' index %s %s %s ' % (x, y, z))
4
5
6  def home(name):
7      time.time()
8      print('welcome %s to home page' % name)
9      return 1234
10
11
12 def outer(func):
13     def wrapper(*args, **kwargs):
14         start = time.time()
15         res = func(*args, **kwargs)
16         stop = time.time()
17         print(stop - start)
18         return res
19
20     return wrapper
21
22
23 home = outer(home)
24 res = home('zhao')
25 print('返回值:>>>', res)

```

- 语法糖:

在被装饰对象正上方的单独一行写 @装饰器名字

```

1  # 装饰器
2  def timer(func):
3      def wrapper(*args, **kwargs):
4          start = time.time()
5          res = func(*args, **kwargs)
6          stop = time.time()
7          print(stop - start)
8          return res
9
10     return wrapper
11
12
13 @timer # index=timer(index)
14 def index(x, y, z):
15     time.sleep(2)
16     print(' index %s %s %s ' % (x, y, z))

```

```

17
18
19 @timmer # home = timmer(home)
20 def home(name):
21     time.sleep(2)
22     print('welcome %s to home page' % name)
23     return 1234
24
25
26
27 index(x=1,y=2,z=3)
28 home('zhao')

```

- 偷梁换柱

即将原函数名指向的内存地址偷梁换柱,所以应该将wrapper做的跟原函数一样才行

手动的将原函数的属性值赋值给wrapper, 需要一个一个的去加, 太麻烦

```

1  def outter(func):
2
3      def wrapper(*args, **kwargs):
4          #手动的将原函数的属性值赋值给wrapper
5          # 函数名wrapper.__name__ =原函数.__name__
6          # 函数名wrapper.__doc__ = 原函数.__doc__
7          wrapper.__name__ = func.__name__
8          wrapper.__doc__ = func.__doc__
9          res = func(*args, **kwargs)
10         return res
11
12     return wrapper
13
14
15 @outter # index=outter(index)
16 def index(x, y):
17     """
18
19     :param x:
20     :param y:
21     :return:
22     """
23     print('index:', x, y)
24
25
26 index(1, 2)
27 print(index.__name__)
28 print(index.__doc__) # help(index)
29

```

自动的将原函数的所有属性值赋值给wrapper

```

1  from functools import wraps
2  def outter(func):
3      @wraps(func)
4      def wrapper(*args, **kwargs):

```



```

5         #手动的将原函数的属性值赋值给wrapper
6         # 函数名wrapper.__name__ =原函数.__name__
7         # 函数名wrapper.__doc__ = 原函数.__doc__
8         # wrapper.__name__ = func.__name__
9         # wrapper.__doc__ = func.__doc__
10        res = func(*args, **kwargs)
11        return res
12
13    return wrapper
14
15
16    @outter # index=outter(index)
17    def index(x, y):
18        """
19
20        :param x:
21        :param y:
22        :return:
23        """
24        print('index:', x, y)
25
26
27    index(1, 2)
28    print(index.__name__)
29    print(index.__doc__) # help(index)

```

- 无参装饰器模板

```

1    def outter(func):
2        def wrapper(*args,**kwargs):
3            res=func(*args,**kwargs)
4            return res
5        return wrapper()

```

- 统计时间的装饰器

```

1    def timer(func):
2        def wrapper(*args,**kwargs):
3            start=time.time()
4            res=func(*args,**kwargs)
5            end=time.time()
6            print(end-start)
7            return res
8        return wrapper()

```

- 认证功能

```

1    def auth(func):
2        def wrapper(*args, **kwargs):
3            name = input('your name :').strip()
4            passwd = input('your password:').strip()
5            if name == 'zhao' and passwd == '132':

```

```

6         res = func(*args, **kwargs)
7         return res
8     else:
9         print("your name or your password is error")
10
11     return wrapper
12
13
14 @auth
15 def index():
16     print('from index')
17
18
19 index()
20

```

- 有参装饰器模板

```

1 def 有参装饰器(x,y,z)
2     def outter(func):
3         def wrapper(*args,**kwargs):
4             res=func(*args,**kwargs)
5             return res
6         return wrapper()
7 @有参装饰器(1,y=2,z=3)
8 def 被装饰对象():
9     pass

```

- 认证功能改进

```

1 def auth(db_type):
2     def deco(func):
3         def wrapper(*args, **kwargs):
4             username = input('your name:').strip()
5             password = input('your paddword').strip()
6             if db_type == 'file':
7                 print('基于文件验证')
8                 if username == 'zhao' and password == '133':
9                     print('login successful')
10                    res = func(*args, **kwargs)
11                    return res
12                else:
13                    print('username or password is error')
14            elif db_type == 'mysql':
15                print("基于数据库")
16            elif db_type == 'ldap':
17                print("基于ldap")
18            else:
19                print('不支持该db_type')
20
21            return wrapper
22        return deco
23
24

```

```

25 @auth(db_type='file')#@deco #index=dexo(index)
26 def index(x, y):
27     print('index:>>>%s %s' % (x, y))
28
29
30 @auth(db_type='mysql')
31 def home(name):
32     print('home :>>>%s' % name)
33
34
35 @auth(db_type='ldap')
36 def transfer():
37     print("transfer:>>>%s" % transfer)
38
39
40 index(1, 2)
41 home('zhao')
42 transfer()

```

- 叠加多个装饰器分析

```

1  def deco1(func1): # func1=wrapper2的内存地址
2      def wrapper1(*args, **kwargs):
3          print('deco1.wrapper1')
4          res1 = func1(*args, **kwargs)
5          return res1
6
7      return wrapper1
8
9
10 def deco2(func2): # func2=wrapper3的内存地址
11     def wrapper2(*args, **kwargs):
12         print('deco1.wrapper2')
13         res2 = func2(*args, **kwargs)
14         return res2
15
16     return wrapper2
17
18
19 def deco3(x):
20     def outter(func3): # func3=被装饰对象index函数的内存地址
21         def wrapper3(*args, **kwargs):
22             print('deco3.outter.wrapper3')
23             res3 = func3(*args, **kwargs)
24             return res3
25
26         return wrapper3
27
28     return outter
29
30
31 # 加载顺序：自下而上
32 @deco1 # index=deco1(wrapper2的内存地址) ===》index=wrapper1的内存地址
33 @deco2 # index=deco2(wrapper3的内存地址) ===》index=wrapper2的内存地址
34 @deco3(11) # @outter==>@index=outter(index)==>index=wrapper3的内存地址

```

```

35 def index(x, y):
36     print('from index %s %s' % (x, y))
37
38
39 print(index)
40
41 # 执行顺序：自上而下即 wrapper1>wrapper2>wrapper3
42 #
43 index(1, 2) # wrapper1(1,2)
44

```

## 4.10、迭代器

- 迭代器即用来迭代取值的工具，而迭代是重复反馈过程的活动，其目的通常是为了逼近所需的目标或结果，每一次对过程的重复称为一次“迭代”，而每一次迭代得到的结果会作为下一次迭代的初始值，单纯的重复并不是迭代
- 迭代器是用来迭代取值的工具，而涉及到把多个值循环取出来的类型有：列表、字符串、元组、字典、集合、打开文件

```

1 l = ['zhao', 'lisi', 'zhangsan ']
2 i = 0
3 while i < len(l):
4     print(l[i])
5     i += 1
6 """此迭代取值的方式只适用于有索引的数据类型，列表，字符串....."""

```

### 而迭代器是不依赖于索引的

**可迭代对象：**但凡内置有 `__iter__` 的方法的都称之为可迭代对象

通过索引的方式进行迭代取值，实现简单，但仅适用于序列类型：字符串，列表，元组。对于没有索引的字典、集合等非序列类型，必须找到一种不依赖索引来进行迭代取值的方式，这就用到了迭代器。

要想了解迭代器为何物，必须事先搞清楚一个很重要的概念：可迭代对象(Iterable)。从语法形式上讲，内置有 `iter` 方法的对象都是可迭代对象，字符串、列表、元组、字典、集合、打开的文件都是可迭代对象：

可迭代对象.`__iter__()` :得到迭代器对象

```

1 str1=''
2 #str1.__iter__()
3 l=[]
4 #l.__iter__()
5 t=(1,)
6 #t.__iter__()
7 d={'a':1}
8 #d.__iter__()
9 set1={1,2,3}
10 #set1.__iter__()
11 with open('a.txt','w') as f:
12     # f.__iter__()

```

```
13     pass
14
```

迭代器对象：内置有 `__next__` 方法并且内置有 `__iter__` 方法的对象

目前只有文件既是可迭代对象，又是迭代器对象

迭代器对象.`__next__()`:得到迭代器的下一个值

迭代器对象.`__iter__()`:得到迭代器的本身，说白了调了就跟没调一样

```
1 d = {'a': 1, 'b': 2, 'c': 3}
2 d_iterator = d.__iter__()
3 print(d_iterator.__next__())
4 print(d_iterator.__iter__())
5 print(d_iterator is d_iterator.__iter__())#True
6
```

## while循环

```
1 d = {'a': 1, 'b': 2, 'c': 3}
2 d_iterator = d.__iter__()
3 while True:
4     try:
5         print(d_iterator.__next__())
6     except StopIteration :
7         break
8
```

## for循环工作原理

for循环又称为迭代循环，in后可以跟任意可迭代对象

```
1 d = {'a': 1, 'b': 2, 'c': 3}
2 d_iterator = d.__iter__()
3
4 for i in d:#d本身就是可迭代对象，调用d.__iter__()后返回的还是它本身
5     print(i)
```

1. `d.__iter__()` 得到一个迭代器对象
2. 迭代器对象.`__next__()` 拿到一个返回值，然后该返回值赋值给i
3. 循环往复步骤2,直到抛出`StopIteration`异常for循环会捕捉异常然后结束迭代。

可迭代对象不是迭代器对象

迭代器对象是可迭代对象

## 4.11、生成器

生成器就是迭代器

```
1 def fun():
2     print('第一次')
3     yield 1
4     print('第二次')
```

```

5     yield 2
6 g=fun()
7 print(g)
8 print(g.__iter__())
9 #print(g.__next__())#会触发函数体代码的运行，然后遇到yield停下来，将yield的值当作本次调用的结果返回
10
11 #res1=g.__next__()
12 res1=next(g)
13 print(res1)
14 res2=g.__next__()
15 print(res2)
16 """结果如下"""
17 <generator object fun at 0x000001c1f8231e00>
18 <generator object fun at 0x000001c1f8231e00>
19 第一次
20 1
21 第二次
22 2
23 """没值 就报StopIteration"""

```

```

1 next(g) 等同于g.__next__()
2 iter(可迭代对象) 等同于 可迭代对象.__iter__()
3

```

## yield表达式 (挂起)

```

1 # x=yield 返回值
2
3 def dog(name):
4     print("dog %s eat something...." % name)
5     while True:
6         # x拿到的时yield接收到的值
7         x = yield # x='一根骨头'
8         print('dog %s eat %s' % (name, x))
9
10
11 g = dog('dH')
12 # print(g)
13
14 # res=next(g)
15 # print(res)
16 # next(g)
17 g.send(None) # 等同于next(g)
18 g.send('一根骨头') # 为yield赋值，yield会将send过来的值交给x
19 # g.send('包子')
20 g.send(['包子', '骨头'])
21 # g.close()#关闭后，无法传值
22

```

```

1 # x=yield 返回值
2

```

```

3  def dog(name):
4      foodlist=[]
5      print("dog %s eat something...." % name)
6      while True:
7          # x拿到的时yield接收到的值
8          x = yield foodlist
9          print('dog %s eat %s' % (name, x))
10         foodlist.append(x)
11
12
13  g = dog('dH')
14  g.send(None)
15  res=g.send('骨头')
16  print(res)
17  res2=g.send('包子')
18  print(res2)
19
20  """结果如下"""
21  dog dH eat something....
22  dog dH eat 骨头
23  ['骨头']
24  dog dH eat 包子
25  ['骨头', '包子']

```

```

1  def func():
2      print('start....')
3      x=yield 1111
4      print('哈哈')
5
6      yield 222
7
8  g=func()
9  res=next(g)
10  print(res)
11
12
13  res3=g.send('xxxxx')
14  print(res3)
15
16  """结果如下"""
17  start....
18  1111
19  哈哈
20  222

```

### 三元表达式

```

1  # 三元表达式
2  # 格式: 条件成立时返回的值 if 条件 else 条件不成立时返回的值
3  z = 1
4  s = 2
5  res = z if z > s else s
6  print(res)
7
8
9  def func(x,y):
10     x=1 if x>y else 2
11     print(x)

```

## 列表生成式

```

1  def func(x, y):
2      x = 1 if x > y else 2
3      print(x)
4
5
6  l = ['zhao', 'lisi', 'wan', 'egon']
7  new_list = []
8  for name in l:
9      if name.endswith('n'):
10         new_list.append(name)
11  # 列表生成式
12  new_l = [name for name in l if name.endswith('n')]
13  print(new_l)
14  print(new_list)
15
16  # 所有小写变成大写
17  res = [name.upper() for name in l]
18  print(res)
19  # 把所有的名字去掉后缀n
20  res2=[ name.replace('n','') for name in l]
21  print(res2)
22
23  """结果如下"""
24  ['wan', 'egon']
25  ['wan', 'egon']
26  ['ZHAO', 'LISI', 'WAN', 'EGON']
27  ['zhao', 'lisi', 'wa', 'ego']

```

## 字典生成式



```

1 keys={'name','age','hobby'}
2 dic={key:None for key in keys}
3 print(dic)
4
5 items=[('name','zhao'),('age',18),('hobby','JayChou')]
6 res={key:value for key,value in items if key!='hobby'}
7 print(res)
8
9 """结果如下"""
10 {'age': None, 'name': None, 'hobby': None}
11 {'name': 'zhao', 'age': 18}

```

## 集合生成式

```

1 keys=['name','age','gender']
2 set2={ name for name in keys}
3 print(set2,type(set2))
4
5 """结果如下"""
6 {'name', 'age', 'gender'} <class 'set'>
7

```

## 生成器表达式

```

1 g=(i for i in range(10) if i>3)
2 print(g,type(g))
3 print(next(g))
4 print(next(g))
5 print(next(g))
6 print(next(g))
7 print(next(g))
8
9 """结果如下"""
10 <generator object <genexpr> at 0x0000012FBFEC0430> <class 'generator'>
11 4
12 5
13 6
14 7
15 8

```

## 4.12、二分法

```

1 # 有一个按照从小到大顺序排列的数字列表，需要从中找到一个想要的数字，怎么做更高效
2 nums = [-3, 1, 10, 4, 77, 8, 13, 21, 43, 89]
3 find_mum = 13
4
5
6 # for num in nums:
7 #     if num ==find_mum:
8 #         print('find it')
9 #         break
10
11 # 二分法

```

```

12
13 def binary_search(find_num, l):
14     print(l)
15     if len(l)==0:
16         print('找的值不存在')
17         return
18     # 找到列表的中间值:
19     mid_index = len(l) // 2
20     mid_val = l[mid_index]
21     if find_num > mid_val:
22         # 接下来应该找列表的右半部分
23         l=l[mid_index+1:]
24         binary_search(find_num, l)
25     elif find_num < mid_val:
26         # 接下来应该找列表的左半部分
27         l=l[:mid_index]
28         binary_search(find_num, l)
29     else:
30         print('find it ')
31     binary_search(find_num, l)

```

## 4.13、面向过程与函数式

### 面向过程

“面向过程”核心是“过程”二字，“过程”指的是解决问题的步骤，即先干什么再干什么……，基于面向过程开发程序就好比在设计一条流水线，是一种机械式的思维方式，这正好契合计算机的运行原理：任何程序的执行最终都需要转换成cpu的指令流水按过程调度执行，即无论采用什么语言、无论依据何种编程范式设计出的程序，最终的执行都是过程式的。

#### 1、优点

- 1 将复杂的问题流程化，进而简单化

#### 2、缺点

- 1 程序的可扩展性极差，

#### 3、应用场景

- 1 面向过程的程序设计一般用于那些功能一旦实现之后就很少需要改变的场景，如果你只是写一些简单的脚本，去做一些一次性任务，用面向过程去实现是极好的，但如果你要处理的任务是复杂的，且需要不断迭代和维护，那还是用面向对象最为方便。

### 函数式

函数式编程并非用函数编程这么简单，而是将计算机的运算视为数学意义上的运算，比起面向过程，函数式更加注重的是执行结果而非执行的过程，代表语言有：Haskell、Erlang。而python并不是一门函数式编程语言，但是仍为我们提供了很多函数式编程好的特性，如lambda, map, reduce, filter

## 匿名函数lambda

对比使用def关键字创建的是有名字的函数，使用lambda关键字创建则是没有名字的函数，即匿名函数，语法如下

```
1 | lambda 参数1,参数2,...: expression
```

案例：

```
1 | # 1、定义
2 | lambda x,y,z:x+y+z
3 |
4 | #等同于
5 | def func(x,y,z):
6 |     return x+y+z
7 |
8 | # 2、调用
9 | # 方式一：
10 | res=(lambda x,y,z:x+y+z)(1,2,3)
11 |
12 | # 方式二：
13 | func=lambda x,y,z:x+y+z # “匿名”的本质就是要没有名字，所以此处为匿名函数指定名字是没有意义的
14 | res=func(1,2,3)
15 | print(res)
```

匿名函数与有名函数有相同的作用域，但是匿名意味着引用计数为0，使用一次就释放，所以匿名函数用于临时使用一次的场景，匿名函数通常与其他函数配合使用

案例

```
1 | salaries={
2 |     'siry':3000,
3 |     'tom':7000,
4 |     'lili':10000,
5 |     'jack':2000
6 | }
```

要想取得薪水的最大值和最小值，我们可以使用内置函数max和min（为了方便开发，python解释器已经为我们定义好了一系列常用的功能，称之为内置的函数，我们只需要拿来使用即可）

```
1 | print(max(salaries))
2 | print(min(salaries))
3 | #默认根据字符比较大小
4 |
5 |
6 | """结果如下"""
7 | tom
8 | jack
```

内置max和min都支持迭代器协议，工作原理都是迭代字典，取得是字典的键，因而比较的是键的最大值和最小值，而我们想要的是比较值的最大值与最小值，于是做出如下改动

```

1 max1=max(salaries,key=lambda k:salaries[k])
2 print(max1)
3 min1=min(salaries,key=lambda k:salaries[k])
4 print(min1)
5
6 """结果如下"""
7 lili
8 jack

```

直接对字典进行排序，默认也是按照字典的键去排序的

```

1 print(sorted(salaries))
2
3 """结果如下"""
4 ['jack', 'lili', 'siry', 'tom']

```

根据值的最大值与最小值排序

```

1 res=sorted(salaries,key=lambda k:salaries[k])
2 print(res)
3
4 """结果如下"""
5 ['jack', 'siry', 'tom', 'lili']

```

- 特点

- 使用lambda关键字创建函数
- 没有名字的函数
- 匿名函数冒号后面的表达式只有一个，注意：是表达式，而不是语句
- 匿名函数自带return，而这个return的结果是表达式计算后的结果

- 缺点

- lambda只能是单个表达式：不是一个代码块，lambda的设计就是为了满足简单的函数场景
- 仅仅能封装有限的逻辑，复杂逻辑实现不了，必须使用def来处理

```

1 #匿名函数
2 m=lambda x,y:x+y
3 #通过变量去调用匿名函数
4 print(m(23,19))
5 M=lambda a,b,c:a*b*c
6 print(M(1,2,3))
7
8 age =15
9 print('可以继续参军, 'if age>18 else'继续上学')
10 C=lambda x,y:x if x>y else y
11 print(C(1,5))
12
13 re=(lambda x,y:x if x<y else y)(16,12)
14 print(re)
15
16 RS=lambda x:(x**2)+890
17 print(RS(10))

```

## map、filter、reduce (了解)

```
1 l = ['zhoa', 'lisi', 'wangwu ']  
2  
3 res = map(lambda name: name + '_dsb', l)  
4 print(res) # 生成器  
5 res1 = filter(lambda name: name.endswith('u'), l)  
6 print(res1)  
7  
8 from functools import reduce  
9  
10 res3=reduce(lambda x, y: x + y, [1, 2, 3],10)  
11 print(res3)  
12  
13 """结果如下"""  
14 <map object at 0x0000014D2CDF7E80>  
15 <filter object at 0x0000014D2CDF7D60>  
16 16
```

## 4.13、函数的递归调用

函数不仅可以嵌套定义，还可以嵌套调用，即在调用一个函数的过程中，函数内部又调用另一个函数，而函数的递归调用指的是在调用一个函数的过程中又直接或间接地调用该函数本身

例如：

在调用f1的过程中，又调用f1，这就是直接调用函数f1本身

```
1 def f1():  
2     print('from f1')  
3     f1()  
4 f1()
```

**递归 自己调用自己**

**必须有一个明确的结束条件**

- 阶乘/通过递归实现

```
1 #阶乘/通过递归实现  
2 def factorial(n):  
3     '''  
4     阶乘  
5     '''  
6     if n==1:  
7         return 1  
8     return n*factorial(n-1)  
9 result =factorial(5)  
10 print(result)
```

- 通过普通函数实现阶乘

```

1  #通过普通函数实现阶乘
2  def jiecheng(n):
3      '''
4      阶乘
5      '''
6      result=1
7      for item in range(1,n+1):
8          result*=item
9      return result
10 re=jiecheng(5)
11 print(re)

```

```

1  #模拟实现，树形结构的遍历
2  import os #引入文件操作模块
3  def findFile(file_Path):
4      listRs=os.listdir(file_Path)#得到该路径下面的文件夹
5      for fileItem in listRs:
6          full_Path=os.path.join(file_Path,fileItem)#获取完整的文件路径
7          if os.path.isdir(full_Path):#判断是否是文件夹
8              findFile(full_Path)#如果是一个文件，再次去遍历
9          else:
10             print(fileItem)
11     else:
12         return
13 #d调用搜索文件对象
14 findFile('D:\\Python')

```

**回溯：**一层 一层调用下去

**递推：**满足某种结束条件，结束递归调用，然后一层一层返回

#### 4.14、内置函数 数学运算

```

1  #Python语言自带的函数
2  print(abs(-23))#取绝对值
3  print(round(2.23))#近似值
4  print(round(12.56,1))#保留一位小数
5  print(pow(3,3))#次方
6  print(3**3)#次方
7  print(max(12,15,18))#返回最大值
8  print(min(12,15))#最小值
9
10 print(sum(range(50)))
11
12 #eval 执行表达式
13 a,b,c=1,2,3
14 print('动态生成的函数{}'.format(eval('a+b+c'))))
15

```

## 4.115、类型转换函数

```
1  #chr() 数字转字符
2  #ord() 字符转数字
3  #bin()  转为二进制
4  #hex()  转为十六进制
5  #oct()  转八进制
6  #list()将元组转列表
7  print(bin(10))
8  print(hex(16))
9  tupleA=(132,2,2,23,1)
10 print(list(tupleA))
11
12 listA=[1,123,15,'ghdj']
13 print(tuple(listA))
14
15 # bytes转换
16 print(bytes('我喜欢Python',encoding='utf-8'))
17
```

## 4.16、序列操作函数

```
1  #sorted()函数对所有可迭代的对象进行排序操作    生成一个新的进行排序
2  #sort()  在原有数据基础上进行排序
3  #all()
4  #range()
5
6  list=[1,2,3,45,6]
7  # list.sort()#直接修改原始对象
8
9  print('-----排序之前-----{}'.format(list))
10 # varList=sorted(list)
11 varList=sorted(list,reverse=True)#降序排序
12 # varList=sorted(list,reverse=False)#升序排序
13 print('-----排序之后-----{}'.format(varList))
14
```

# 05模块与包

### 模块是个啥

模块是一系列功能的集合体，分为三大类：

1. 内置模块
2. 第三方模块
3. 自定义的模块

一个Python文件本身就是一个模块，文件名m.py，模块名叫m

`__init__.py` 文件，称为包

### 为何有模块

1. 内置模块与第三方拿来就用，无需定义，可以提升开发效率
2. 自定义模块，可以将程序的各部分功能提取出来放到一个模块中，供大家共享，好处是减少代码冗余，程序组织结构清晰

## 5.1、首次导入模块发生的三件事

- 1. 执行xx.py
- 2. 产生xx.py的名称空间，将xx.py运行过程中产生的名字都丢带xx的名称空间中
- 3. 在当前文件中产生的有一个名字xx,该名字指向2中产生的名称空间

首次导入完后，之后再导入都是直接引用首次导入产生的xx.py的名称空间,不会重复执行代码

## 5.2、import导入模块使用

```
1  """
2
3  1. 可以使用逗号分隔符在一行导入多个模块
4  import time,sys
5  2. 导入模块的规范（导入的顺序）
6      1.python内置的模块
7      2.第三方模块
8      3.自定义模块
9  3. 将导入的模块起个别名(当模块名比较长的时候用as起别名)
10 import time as t
11 4. 自定义模块命名应该采用纯小写+下划线风格
12 5. 可以在函数内导入模块，对比在文件开头导入模块属于全局作用域，在函数内导入的模块则属于局部的作用域。
13 """
```

### 一个python文件的两种用途

- 被当作模块导入
- 被当作程序运行

一个Python文件有两种用途，一种被当主程序/脚本执行，另一种被当模块导入，为了区别同一个文件的不同用途，每个py文件都内置了 `__name__` 变量，该变量在py文件被当做脚本执行时赋值为“main”,在py文件被当做模块导入时赋值为模块名

```
1  print(__name__)
2  #1. 当xx.py被运行时，__name__的值为__main__
3  #2. 当xx.py被当作模块导入时，__name__的值为'xx'
4
5  if __name__ == '__main__':
6      print("文件被执行")
7  else:
8      #被当做模块导入的时做到事情
9      print("文件被导入")
```



## 5.3、from ...import 导入

```
1  """
2  import导入模块在使用时必须加前缀"模块."
3      优点：肯定不会与当前名称空间中的名字冲突
4      缺点：加前缀显得麻烦
5
6  from ...import ...导入也发生了三件事
7      1.产生一个模块的名称空间
8      2.运行xx.py将运行过程中产生的名字都丢到模块的名称空间去
9      3. 在当前名称空间拿到一个名字，该名字与模块名称空间中的某一个内存地址
10     优点：代码更精简
11     缺点：容易与当前名称空间混淆
12
13     一行可以导入多个名字
14     from time import time,sleep
15     *: 导入模块中的所有名字
16     from os import *
17
18     """
```

如果我们需要引用模块中的名字过多的话，可以采用上述的导入形式来达到节省代码量的效果，但是需要强调的一点是：只能在模块最顶层使用的方式导入，在函数内则非法，并且的方式会带来一种副作用，即我们无法搞清楚究竟从源文件中导入了哪些名字到当前位置，这极有可能与当前位置的名字产生冲突。模块的编写者可以在自己的文件中定义 `__all__` 变量用来控制\*代表的意思

```
1  #foo.py
2  __all__=['x','get'] #该列表中所有的元素必须是字符串类型，每个元素对应foo.py中的一个名字
3  x=1
4  def get():
5      print(x)
6  def change():
7      global x
8      x=0
9  class Foo:
10     def func(self):
11         print('from the func')
```

这样我们在另外一个文件中使用\*导入时，就只能导入 `__all__` 定义的名字了

```
1  from foo import * #此时的*只代表x和get
2
3  x #可用
4  get() #可用
5  change() #不可用
6  Foo() #不可用
```

## 5.4、循环导入问题

## 5.5、模块搜索优先级

模块其实分为四个通用类别，分别是：

- 1、使用纯Python代码编写的py文件
- 2、包含一系列模块的包
- 3、使用C编写并链接到Python解释器中的内置模块
- 4、使用C或C++编译的扩展模块

### 优先级

在导入一个模块时，如果该模块已**加载到内存**中，则直接引用，否则会**优先查找内置模块**，然后按照从左到右的顺序**依次检索sys.path中定义的路径**，直到找模块对应的文件为止，否则抛出异常。

**sys.path**也被称为模块的搜索路径，它是一个列表类型

```
1 import sys
2 print(sys.path) #值为一个列表，存放了一系列的文件夹，其中第一个文件夹是当前执行文件所在的文件夹
3
4 """结果如下"""
5 ['E:\\Python\\python\\demo\\day04', 'E:\\Python\\python\\demo',
6  'D:\\Catalog\\JetBrains\\PyCharm
  2022.2\\plugins\\python\\helpers\\pycharm_display',
  'C:\\Users\\31812\\AppData\\Local\\Programs\\Python\\Python310\\python310.zip',
  'C:\\Users\\31812\\AppData\\Local\\Programs\\Python\\Python310\\DLLs',
  'C:\\Users\\31812\\AppData\\Local\\Programs\\Python\\Python310\\lib',
  'C:\\Users\\31812\\AppData\\Local\\Programs\\Python\\Python310',
  'C:\\Users\\31812\\AppData\\Local\\Programs\\Python\\Python310\\lib\\site-
  packages', 'D:\\Catalog\\JetBrains\\PyCharm
  2022.2\\plugins\\python\\helpers\\pycharm_matplotlib_backend']
```

列表中的每个元素其实都可以当作一个目录来看：在列表中会发现有.zip或.egg结尾的文件，二者是不同形式的压缩文件，事实上Python确实支持从一个压缩文件中导入模块，我们也只需要把它们都当成目录去看即可。

...

sys.path中的第一个路径通常为**空**，代表执行文件所在的路径，所以在被导入模块与执行文件在同一目录下时肯定是可以正常导入的，而针对被导入的模块与执行文件在不同路径下的情况，为了确保模块对应的源文件仍可以被找到，需要将源文件foo.py所在的路径添加到sys.path中，假设foo.py所在的路为pythoner/projects/

```

1 import sys
2 #找foo.py, 就把foo.py的文件夹添加到环境变量中
3 sys.path.append(r'/pythoner/projects/')
4
5 import foo #无论foo.py在何处,我们都可以导入它了

```

**sys.modules**查看已经加载带内存中的模块

```

1 print('demo1' in sys.modules)

```

## 5.6、代码规范

我们在编写py文件时，需要时刻提醒自己，该文件既是给自己用的，也有可能被其他人使用，因而代码的可读性与易维护性显得十分重要，为此我们在编写一个模块时最好按照统一的规范去编写，如下

```

1 #!/usr/bin/env python #通常只在类unix环境有效,作用是可以使用脚本名来执行,而无需直接调用解释器。
2
3 "The module is used to..." #模块的文档描述
4
5 import sys #导入模块
6
7 x=1 #定义全局变量,如果非必须,则最好使用局部变量,这样可以提高代码的易维护性,并且可以节省内存提高性能
8
9 class Foo: #定义类,并写好类的注释
10     'Class Foo is used to...'
11     pass
12
13 def test(): #定义函数,并写好函数的注释
14     'Function test is used to...'
15     pass
16
17 if __name__ == '__main__': #主程序
18     test() #在被当做脚本执行时,执行此处的代码

```

## 5.7、函数的提示类型

```

1 def register(name:str,age:int,hobby:tuple)->int:# ->int表示返回值为整型,冒号跟的是提示信息
2     print(name)
3     print(age)
4     print(hobby)
5     return 111
6 # register(1,'aaa',[1,])
7 register('lisi',22,('Jaychou',))

```

## 5.8、包

1. 包就是一个包含 `__init__.py` 文件的文件夹
2. 包的本质就是模块的一种形式，是用来被当作模块导入

### 导入包发生的三件事

- 1. 产生一个名称空间
- 2. 运行包下的 `__init__.py` 文件，将运行过程中产生的名字都丢带1的名称空间中
- 3. 在当前执行文件的名称空间中拿到一个名字xx,xx指向1的名称空间

```
1 在python3中，即使包下没有__init__.py文件，import 包仍然不会报错，而在python2中，包下一  
   定要有该文件，否则import 包报错  
2  
3 #2. 创建包的目的是为了运行，而是被导入使用，记住，包只是模块的一种形式而已，包的本质就是一种模
```

### 强调

```
1 1. 关于包相关的导入语句也分为import和from ... import ...两种，但是无论哪种，无论在什么位  
   置，在导入时都必须遵循一个原则：凡是在导入时带点的，点的左边都必须是一个包，否则非法。可以带  
   有一连串的点，如import 顶级包.子包.子模块,但都必须遵循这个原则。但对于导入后，在使用时就没  
   有这种限制了，点的左边可以是包,模块，函数，类(它们都可以用点的方式调用自己的属性)。  
2  
3 2、包A和包B下有同名模块也不会冲突，如A.a与B.a来自俩个命名空间  
4  
5 3、import导入文件时，产生名称空间中的名字来源于文件，import 包，产生的名称空间的名字同样来  
   源于文件，即包下的__init__.py，导入包本质就是在导入该文件
```

针对包内的模块之间互相导入，导入的方式有两种

#### 1、绝对导入：以顶级包为起始

```
1 #demo下的__init__.py  
2 from demo import versions  
3 from demo.a.m1 import b
```

```
1 import也能使用绝对导入，导入过程中同样会依次执行包下的__init__.py,只是基于import导入的结  
   果，使用时必须加上该前缀
```

#### 2、相对导入：.代表当前文件所在的目录，..代表当前目录的上一级目录，依此类推

```
1 #demo下的__init__.py  
2 from . import versions
```

### 总结

包的使用需要牢记三点

- 1、导包就是在导包下`init.py`文件
- 2、包内部的导入应该使用相对导入，相对导入也只能在包内部使用，而且...取上一级不能出包
- 3、使用语句中的点代表的是访问属性  
`m.n.x` ----> 向m要n，向n要x

而导入语句中的点代表的是路径分隔符

import a.b.c --> a/b/c, 文件夹a下有子文件夹b, 文件夹b下有子文件或文件夹c

所以导入语句中点的左边必须是一个包

### from 包 import \*

在使用包时同样支持from pool.futures import \*, 毫无疑问\*代表的是futures下\_\_init\_\_.py中所有的名字, 通用是用变量\_\_all\_\_来控制\*代表的意思

```
1 #futures下的__init__.py
2 __all__=['process', 'thread']
```

## 06、Python面向对象

### 6.1、oop介绍

面向对象编程 oop 【object oriented programming】是一种Python的编程思路

- 面向过程：  
在思考问题的时候，怎么按照步骤去实现，  
然后将问题解决拆分成若干个步骤，并将这些步骤对应成方法一步一步的最终完成功能
- 面向过程：就是一开始学习的，按照解决问题步骤去编写代码【根据业务逻辑去写代码】
- 面向对象：关注的是设计思维【找洗车店，给钱洗车】
- 从计算机角度看：面向过程不适合做大项目
- 面向过程关注的是：怎么做
- 面向对象关注的是：谁来做

### 6.2、类和对象

- 类：是一个模板，模板里包含多个函数，函数里实现一些功能
- 对象：则是根据模板创建的实例，通过实例对象可以执行类中的函数
- 类由3部分构成：
  - 类的名称：类名
  - 类的属性：一组数据
  - 类的方法：允许对进行操作的方法（行为）

例如：创建一个人类

事物的名称（类名）：人（Person）

属性：身高，年龄

方法：吃 跑.....

- 类是具有一组 相同或者相似特征【属性】和行为【方法】的一系列对象的集合

现实世界 计算机世界

行为-----》方法

特征-----》属性

- 对象：是实实在在的一个东西，类的具象化 实例化
- 类是对象的抽象化 而对象是类的实例

```

1  """
2  在程序中，必须要事先定义类，然后再调用类产生对象（调用类拿到的返回值就是对象）。产生对象的类
   与对象之间存在关联，这种关联指的是：对象可以访问到类中共有的数据与功能，所以类中的内容仍然是
   属于对象的，类只不过是一种节省空间、减少代码冗余的机制，面向对象编程最终的核心仍然是去使用对
   象。
3  """

```

## 6.3、定义类

#定义类和对象

#类名：采用大驼峰方式命名

```

1  #创建对象
2  #对象名=类名()
3  '''
4
5  class 类名:
6      属性
7      方法
8
9  '''
10 #实例方法：
11 # 在类的内部，使用def关键字可以定义一个实例方法，与一般函数 定义不同，类方法必须包含参数
   self【self可以是其他的名字，但是这个位置必须被占用】，且为第一个参数
12 #属性：在类的 内部定义的变量
13 #定义在类里面，方法外面的属性成为类属性，定义在方法里面使用self引用的属性称之为实例属性
14 class Person:
15     '''
16     对应人的特征
17     '''
18     name='小勇'    #类属性
19     age=22         #类属性
20     '''
21     对应人的行为
22     '''
23
24     def __init__(self):
25         self.name = '小赵'#实例属性
26     def eat(self):#实例方法
27         print('狼吞虎咽的吃')
28     def run(self):#实例方法
29         print('飞快地跑')
30
31
32

```

类体最常见的是变量的定义和函数的定义，但其实类体可以包含任意Python代码，类体的代码在类定义阶段就会执行，因而会产生新的名称空间用来存放类中定义的名字，可以打印 `Person.__dict__` 来查看类这个容器内盛放的东西

```

1 print(Person.__dict__)
2
3 """结果如下"""
4 {'__module__': '__main__', '__doc__': '\n    对应人的特征\n    ', 'name': '小
  勇', 'age': 22, '__int__': <function Person.__int__ at 0x000001E768DCFE0>,
  'eat': <function Person.eat at 0x000001E76907A950>, 'run': <function
  Person.run at 0x000001E77A2FA9E0>, '__dict__': <attribute '__dict__' of
  'Person' objects>, '__weakref__': <attribute '__weakref__' of 'Person'
  objects>}
5

```

调用类的过程称为将类实例化，拿到的返回值就是程序中的对象，或称为一个实例

```

1 #创建对象【类的实例化】
2 xm=Person() # 每实例化一次Person类就得到一个人的对象
3 xm.eat()#调用函数
4 xm.run()
5 print('{}的年龄是{}'.format(xm.name,xm.age))

```

## 6.4、init方法

```

1 # 如果有n个这样对象 被实例化，那么就需要添加很多次实例属性，显然比较麻烦
2 class Person1:
3     student='周杰伦'
4     def __init__(self,name,age,sex):#魔术方法
5         '''
6         实例属性的声明
7         '''
8         self.name=name
9         self.age=age
10        self.sex=sex
11    def run(self,name):
12        print('%s 跑太快了吧'%self.name)
13

```

然后实例出三个人

```

1 per1=Person1('zhao',18,'男')
2 per=Person1('lisi',22,'女')
3 per=Person1('wangwu',19,'男')
4 print(per1.age)#18

```

单拿per1的产生过程来分析，调用类会先产生一个空对象per1，然后将per1连同调用类时括号内的参数一起传给 Person1.\_\_init\_\_(per1,'李zhao',18,'男')

```

1 def __init__(self, name, age, sex):
2
3     self.name = name # per1.name = 'zhao'
4     self.age = age # per1.age = 18
5     self.sex = sex # per1.sex = '男'

```

会产生对象的名称空间，同样可以用 `__dict__` 查看

```

1 print(per1.__dict__)
2
3 """结果如下"""
4 {'name': 'zhao', 'age': 18, 'sex': '男'}

```

## 6.5、属性访问

### 类属性与对象属性

在类中定义的名字，都是类的属性，细说的话，类有两种属性：数据属性和函数属性，可以通过 `__dict__` 访问属性的值，比如 `Person1.__dict__['student']`，但Python提供了专门的属性访问语法

```

1 print(Person1.student) # 访问数据属性，等同于Person1.__dict__['student']
2 print(Person1.run) # 访问函数属性，等同于Person1.__dict__['run']
3
4 """结果如下"""
5
6 周杰伦
7 <function Person1.run at 0x0000029E9DCDA950>

```

操作对象的属性也是一样

```

1 print(per1.name) # print(per1.__dict__['name'])
2 print(per1.age)
3 res = per1.hobby = 'JayChou' # 新增，等同于
   res=per1.__dict__['hobby']='JayChou'
4 print(res)
5 print(per1.hobby)
6 del per1.hobby # 删除，等同于del per1.__dict__['hobby']
7
8

```

对象的名称空间里只存放着对象独有的属性，而对象们相似的属性是存放于类中的。对象在访问属性时，会优先从对象本身的 `__dict__` 中查找，未找到，则去类的 `__dict__` 中查找

## 6.5、self理解

- self和对象指向同一个地址，可以认为self就是对象的引用
- 在实例化对象时，self不需要开发者传参，Python自动将对象传递给self
- self只有类中定义实例方法的时候才有意义，在调用的时候不必传入相应的参数，而是由解释器自动取指向
- self的名字时可以更改的，可以定义成其他的名字，只是约定俗成的定义成了self
- self指的是类实例对象本身



```

1 class Person:
2     def __init__(self,pro):
3         self.pro=pro
4     def geteat(s,name,food):
5         # print(self)
6         print('self在内存中的地址%s'%(id(s)))
7         print('%s喜欢吃%s,专业是: %s'%(name,food,s.pro))
8
9 zs=Person('心理学')
10 print('zs的内存地址%s'%(id(zs)))
11 zs.geteat('小王','榴莲')

```

## 6.6、魔术方法

魔术方法：\_\_xxx\_\_

```

1 '''
2 __init__ 方法：初始化一个类，在创建实例对象为其赋值时使用
3 __str__方法： 在将对象转换成字符串str(对象)测试的时候，打印对象的信息
4 __new__方法： 创建并返回一个实例对象，调用了一次，就会得到一个对象
5 __class__方法： 获得已知对象的类（对象__class__）
6
7 __del__方法： 对象在程序运行结束后进行对象销毁的时候调用这个方法，来释放资源
8 '''
9
10 class Animal:
11     def __init__(self,name,color):
12         self.name=name
13         self.color=color
14         print('-----init-----')
15
16     def __str__(self):
17         return '我的名字是%s,我的颜色为%s'%(self.name,self.color)
18
19     def __new__(cls, *args, **kwargs):
20         print('-----new-----')
21         return object.__new__(cls)
22 dog =Animal('旺财','黑色')
23 print(dog)
24
25
26 '''
27 __new__和__init__函数区别
28 __new__类的实例化方法：必须返回该实例 否则对象就创建不成功
29 __init__用来做数据属性的初始化工作，也可以认为是实例的构造方法，接收类的实例 self 并对其
    进行构造
30 __new__至少有一个参数是cls是代表要实例化的类，此参数在实例化时由Python解释器自动操作
31 __new__函数执行要早于__init__函数
32 '''
33

```

```

1 class People:
2     def __init__(self, name, age):
3         self.name=name
4         self.age=age
5     def __str__(self):
6         return f'{self.name}今年{self.age}岁'
7
8 p=People('zhao',18)
9 print(p)
10
11 """结果如下"""
12 zhao今年18岁

```

```

1 class People1:
2     def __init__(self, name, age):
3         self.name=name
4         self.age=age
5     def __del__(self):
6         print('run.....')
7 People1('lisi',12)
8 print('=====')
9
10 """结果如下"""
11 run.....
12 =====
13

```

## 6.7、析构方法

```

1 '''
2 当一个对象被删除或者被销毁是，python解释器会默认调用一个方法，这个方法为__del__()方法，也
3 被成为析构方法
4 '''
5 class Animals:
6     def __init__(self, name):
7         self.name=name
8         print('这是构造初始化方法')
9     def __del__(self):
10        print('当在某个作用域下面，没有被引用的情况下，解释器会自动调用此函数，来释放内存
11        空间')
12        print('这是析构方法')
13        print('%s 这个对象被彻底清理了，内存空间被释放了'%self.name)
14
15 cat=Animals('猫猫')
16 # del cat #手动的清理删除对象
17 input('程序等待中.....')
18
19 #当整个程序脚本执行完毕后会自动的调用__del__方法
20 # 当对象被手动摧毁时也会自动调用__del__方法
21 # 析构方法一般用于资源回收，利用__del__方法销毁对象回收内存资源

```

## 6.8、封装

面向对象编程有三大特性：封装、继承、多态，其中最重要的一个特性就是封装。封装指的就是把数据与功能都整合到一起，针对封装到对象或者类中的属性，我们还可以严格控制对它们的访问，分两步实现：**隐藏与开放接口**，也就是后面出现的**私有化属性和方法**

```
1  # -*- coding: UTF-8 -*-
2  # @Date : 2022/9/6 19:17
3
4  class School:
5      school_name = 'ShanXiPoliceCollege'
6
7      def __init__(self, nickname, addr):
8          self.addr = addr
9          self.nickname = nickname
10         self.classes = []
11
12         def related(self, class_obj):
13             # self.classes.append(class_name)
14             self.classes.append(class_obj)
15
16         def tell_class(self):
17             # 打印班级的名字
18             print(self.nickname.center(60, '='))
19             # 打印班级开设的课程信息
20             for class_obj in self.classes:
21                 class_obj.tell_couse()
22
23
24     """
25     # 创建学校
26     sch_obj1 = School('老男孩', '上海')
27     sch_obj2 = School('黑马程序员', '北京')
28
29     # 开设班级
30     sch_obj1.related('脱产14期')
31     sch_obj2.related('脱产15期')
32     #查看每个校区开设的班级
33     sch_obj1.tell_class()
34     sch_obj2.tell_class()
35     # sch_obj1.tell_classes()
36     """
37     sch_obj1 = School('老男孩', '上海')
38     sch_obj2 = School('黑马程序员', '北京')
39     sch_obj3 = School('千锋教育', '北京')
40
41
42     class Class:
43         def __init__(self, name):
44             self.name = name
45             self.course = None
46
47         def related_course(self, course_obj):
48             self.course = course_obj
49
```

```

50     def tell_couse(self):
51         print('班级名:%s' % self.name, end=' ')
52         self.course.tell_info() # 打印课程的详细信息
53
54
55 # 创建班级
56 class_obj1 = Class('脱产14期')
57 class_obj2 = Class('脱产15期')
58 class_obj3 = Class('脱产16期')
59 # 2.为班级关联一个课程
60 # class_obj1.related_course('Python全栈开发')
61 # class_obj2.related_course('Linux运维')
62 # class_obj3.related_course('MySQL数据库')
63
64
65 # 3.查看班级开设的课程
66 # class_obj1.tell_couse()
67 # class_obj2.tell_couse()
68 # class_obj3.tell_couse()
69
70 # 4. 为学校开设班级
71 sch_obj1.related(class_obj1)
72 sch_obj2.related(class_obj2)
73 sch_obj3.related(class_obj2)
74
75
76 # sch_obj2.tell_class()
77 # sch_obj1.tell_class()
78
79
80 class Course:
81     def __init__(self, name, period, price):
82         self.name = name
83         self.period = period
84         self.price = price
85
86     def tell_info(self):
87         print('<%s-%s-%s>' % (self.name, self.period, self.price))
88
89
90 # 三: 课程
91 # 1.创建课程
92 course_obj1 = Course('Python全栈开发', '6months', 20000)
93 course_obj2 = Course('Linux运维', '5months', 15000)
94 course_obj3 = Course('MySQL数据库', '2months', 8000)
95 # 2.查看课程详细信息
96 # course_obj1.tell_info()
97 # course_obj2.tell_info()
98
99 # 3.为班级关联课程对象
100 class_obj1.related_course(course_obj1)
101 class_obj2.related_course(course_obj2)
102 class_obj3.related_course(course_obj3)
103
104 # 4.

```

```

105 # class_obj1.tell_couse()
106 # class_obj2.tell_couse()
107 # class_obj3.tell_couse()
108
109 sch_obj1.tell_class()
110 sch_obj2.tell_class()
111 sch_obj3.tell_class()
112
113
114

```

## 6.9、继承

### Python中展现面向对象的三大类型：封装，继承，多态

1. 封装：值得是把内容封装到某个地方，便于后面的使用  
他需要：  
把内容封装到某个地方，从另外一个地方去调用被封装的内容  
对于封装来说，其实就是使用初始化构造方法将内容封装到对象中，然后通过对象直接或者self来获取被封装的内容
2. 继承：和现实生活当中的继承是一样的，也就是子可以继承父的内容【属性和行为】（爸爸有的儿子有，相反，儿子有的爸爸不一定有）。**继承是一种创建新类的方式，在Python中，新建的类可以继承一个或多个父类，新建的类可称为子类或派生类，父类又可称为基类或超类**
3. 所谓多态，定义时的类型和运行时的类型是不一样，此时就成为多态

#### 单继承

```

1 class Animal:
2     def eat(self):
3         '''
4         吃
5         '''
6         print('吃')
7     def drink(self):
8         '''
9         喝
10        '''
11        print('喝')
12
13 class Dog(Animal):#继承Animal父类，    此时Dog就是子类
14     def wwj(self):
15         print('汪汪叫')
16 class Cat(Animal):
17     def mmj(self):
18         print('喵喵叫')
19 d1=Dog()
20 d1.eat()#继承了父类的行为
21 d1.wwj()
22 c1=Cat()
23 c1.drink()
24 c1.eat()
25
26 '''
27 对于面向对象的继承来说，其实就是将多个子类共有的方法提取到父类中，

```

```
28 子类仅仅需要继承父类而不必一一去实现
29 这样就可以极大提高效率，减少代码的重复编写，精简代码的层级结构 便于扩展
30
31 class 类名(父类):
32     pass
33     '''
```

通过类的内置属性 `__bases__` 可以查看类继承的所有父类

```
1 print(Dog.__bases__)
2
3 """结果如下"""
4 (<class '__main__.Animal'>,)
5
```

## 多继承

```
1 class shenxian:
2     def fly(self):
3         print('神仙会飞')
4
5 class Monkey:
6     def chitao(self):
7         print('猴子喜欢吃桃子')
8 class SunWuKong(shenxian,Monkey):
9     pass
10
11 swk=SunWuKong()
12 swk.fly()
13 swk.chitao()
14
15 #当多个父类当中存在相同方法时候,
16 class D:
17     def eat(self):
18         print('D.eat')
19 class C(D):
20     def eat(self):
21         print('C.eat')
22 class B(D):
23     pass
24 class A(B,C):
25     pass
26
27 a=A()
28 # b=B()
29 a.eat()
30 print(A.__mro__)#可以现实类的依次继承关系      查找执行顺序
31 #在执行eat方法时，顺序应该是
32 #首先到A里面去找，如果A中没有，则就继续去B类中去查找，如果B中没有，则去C中查找，
33 #如果C类中没有，则去D类中查找，如果还没有找到，就会报错
34 #A-B-C-D 也是继承的顺序
35
```

在Python2中有经典类与新式类之分，没有显式地继承object类的类，以及该类的子类，都是经典类，显式地继承object的类，以及该类的子类，都是新式类。而在Python3中，即使没有显式地继承object，也会默认继承该类，如下

```
1 print(shenxian.__bases__)
2
3 """结果如下"""
4 (<class 'object'>,)
5
```

1 提示: object类提供了一些常用内置方法的实现，如用来在打印对象时返回字符串的内置方法\_\_str\_\_

## 属性查找

有了继承关系，对象在查找属性时，先从对象自己的\_\_dict\_\_中找，如果没有则去子类中找，然后再去父类中找.....

```
1 >>> class Foo:
2 ...     def f1(self):
3 ...         print('Foo.f1')
4 ...     def f2(self):
5 ...         print('Foo.f2')
6 ...         self.f1()
7 ...
8 >>> class Bar(Foo):
9 ...     def f1(self):
10 ...         print('Foo.f1')
11 ...
12 >>> b=Bar()
13 >>> b.f2()
14 Foo.f2
15 Foo.f1
16 """
17 b.f2()会在父类Foo中找到f2，先打印Foo.f2，然后执行到self.f1()，即b.f1()，仍会按照：对象
18 本身->类Bar->父类Foo的顺序依次找下去，在类Bar中找到f1，因而打印结果为Foo.f1
"""
```

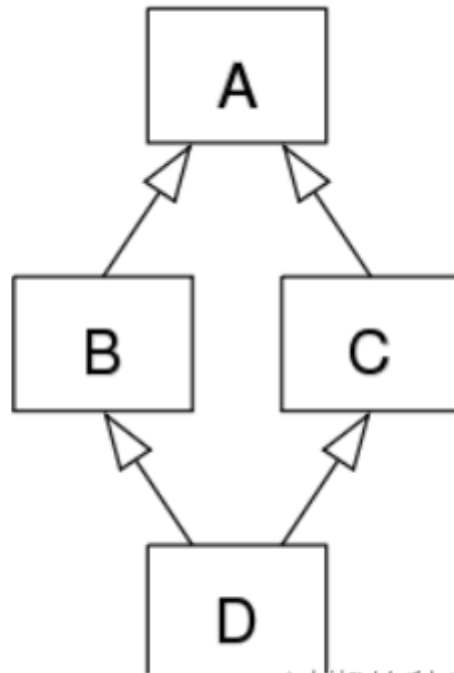
父类如果不想让子类覆盖自己的方法，可以采用双下划线开头的方式将方法设置为私有的

```
1 >>> class Foo:
2 ...     def __f1(self): # 变形为_Foo__f1
3 ...         print('Foo.f1')
4 ...     def f2(self):
5 ...         print('Foo.f2')
6 ...         self.__f1() # 变形为self._Foo__f1,因而只会调用自己所在的类中的方法
7 ...
8 >>> class Bar(Foo):
9 ...     def __f1(self): # 变形为_Bar__f1
10 ...         print('Foo.f1')
11 ...
12 >>>
13 >>> b=Bar()
14 >>> b.f2() #在父类中找到f2方法，进而调用b._Foo__f1()方法，同样是在父类中找到该方法
```

```
15 | Foo.f2
16 | Foo.f1
```

## 继承实现原理

1. 菱形问题,或称钻石问题, 有时候也被称为“死亡钻石”



CSDN@过期的秋刀鱼-\_-

```
1 | class A(object):
2 |     def test(self):
3 |         print('from A')
4 |
5 |
6 | class B(A):
7 |     def test(self):
8 |         print('from B')
9 |
10 |
11 | class C(A):
12 |     def test(self):
13 |         print('from C')
14 |
15 |
16 | class D(B,C):
17 |     pass
18 |
19 |
20 | obj = D()
21 | obj.test() # 结果为: from B
```

## 2. 继承原理

python到底是如何实现继承的呢? 对于你定义的每一个类, Python都会计算出一个方法解析顺序(MRO)列表, 该MRO列表就是一个简单的所有基类的线性顺序列表, 如下



```

1 >>> D.mro() # 新式类内置了mro方法可以查看线性列表的内容，经典类没有该内置方法
2 [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
  '__main__.A'>, <class 'object'>]

```

python会在MRO列表上从左到右开始查找基类,直到找到第一个匹配这个属性的类为止。

MRO列表并遵循如下三条准则:

- 1 1. 子类会先于父类被检查
- 2 2. 多个父类会根据它们在列表中的顺序被检查
- 3 3. 如果对下一个类存在两个合法的选择,选择第一个父类

所以obj.test()的查找顺序是，先从对象obj本身的属性里找方法test，没有找到，则参照属性查找的发起者(即obj)所处类D的MRO列表来依次检索，首先在类D中未找到，然后再B中找到方法test

- 1 1. 由对象发起的属性查找，会从对象自身的属性里检索，没有则会按照对象的类.mro()规定的顺序依次找下去，
- 2 2. 由类发起的属性查找，会按照当前类.mro()规定的顺序依次找下去，

## 6.10、重写父类方法

- 所谓重写，就是子类中，有一个和父类相同名的方法，在子类中的方法会覆盖掉子类中同名的方法，
- 为什么要重写，父类的方法已经不满足子类的需要，那么子类就可以重写父类或者完善父类方法

当父类的方法实现不能满足子类的时候，可以对方法进行重写

重写父类的方法有两种

### 1. 覆盖父类方法

```

1 class Father:
2     def smoke(self):
3         print('抽芙蓉王')
4     def drink(self):
5         print('喝二锅头')
6 class Son(Father):
7     #与父类的（抽烟）方法同名,这就是重写父类方法
8     def smoke(self):
9         print('抽中华')
10
11 #重写父类方法后，子类调用父类方法时，将调用的是子类的方法
12 son=Son()
13 son.smoke()
14

```

### 2. 扩展父类方法

方法一：“指名道姓”地调用某一个类的函数

```

1 class Father:
2     def smoke(self):

```

```

3         print('抽芙蓉王')
4
5     def drink(self):
6         print('喝二锅头')
7
8
9     class Son(Father):
10        def drink(self):
11            ##调用的是函数,因而需要传入self
12            Father.drink(self)
13
14
15    s = Son()
16    s.drink()

```

## 方法二: super()

```

1     """
2     调用super()会得到一个特殊的对象，该对象专门用来引用父类的属性，且严格按照MRO规定的顺序向后
    查找
3     """

```

```

1     class Father:
2         def smoke(self):
3             print('抽芙蓉王')
4
5         def drink(self):
6             print('喝二锅头')
7
8
9     class Son(Father):
10        def drink(self):
11            #调用的是绑定方法，自动传入self
12            super().drink()
13
14
15    s = Son()
16    s.drink()
17
18    """结果如下"""
19    喝二锅头

```

```

1     class People:
2         school = '清华大学'
3
4         def __init__(self, name, sex, age):
5             self.name = name
6             self.sex = sex
7             self.age = age
8
9
10    class Teacher(People):
11        def __init__(self, name, sex, age, title): # 派生
12            #方法一

```

```

13     People.__init__(self, name, sex, age) # 调用的是函数,因而需要传入self
14     #方法二
15     super().__init__(name,age,sex) #调用的是绑定方法,自动传入self
16     self.title = title
17     def teach(self):
18         print('%s is teaching' % self.name)
19 obj = Teacher('lili', 'female', 28, '高级讲师') # 只会找自己类中的__init__,并不会自动调用父类的
20 print(obj.name)

```

这两种方式的区别是：方式一是跟继承没有关系的，而方式二的super()是依赖于继承的，并且即使没有直接继承关系，super()仍然会按照MRO继续往后查找

```

1  #A没有继承B
2
3
4  class A:
5      def test(self):
6          super().test()
7
8
9  class B:
10     def test(self):
11         print('from B')
12
13
14  class C(A, B):
15     pass
16
17  print(C.mro()) # 在代码层面A并不是B的子类,但从MRO列表来看,属性查找时,就是按照顺序C->A->B->object, B就相当于A的“父类”
18
19  obj = C()
20  obj.test() # 属性查找的发起者是类C的对象obj,所以中途发生的属性查找都是参照C.mro()
21
22  """结果如下"""
23  [<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
24  from B

```

obj.test()首先找到A下的test方法，执行super().test()会基于MRO列表(以C.mro()为准)当前所处的位置继续往后查找(),然后在B中找到了test方法并执行。

## 6.11、组合

在一个类中以另外一个类的对象作为数据属性，称为类的组合。组合与继承都是用来解决代码的重用性问题。不同的是：继承是一种“是”的关系，比如老师是人、学生是人，当类之间有很多相同的之处，应该使用继承；而组合则是一种“有”的关系，比如老师有生日，老师有多门课程，当类之间有显著不同，并且较小的类是较大的类所需要的组件时，应该使用组合，如下示例

```

1  class Course:
2      def __init__(self,name,period,price):
3          self.name=name
4          self.period=period

```

```

5         self.price=price
6     def tell_info(self):
7         print('<%s %s %s>' %(self.name,self.period,self.price))
8
9     class Date:
10        def __init__(self,year,mon,day):
11            self.year=year
12            self.mon=mon
13            self.day=day
14        def tell_birth(self):
15            print('<%s-%s-%s>' %(self.year,self.mon,self.day))
16
17    class People:
18        school='清华大学'
19        def __init__(self,name,sex,age):
20            self.name=name
21            self.sex=sex
22            self.age=age
23
24    #Teacher类基于继承来重用People的代码，基于组合来重用Date类和Course类的代码
25    class Teacher(People): #老师是人
26        def __init__(self,name,sex,age,title,year,mon,day):
27            super().__init__(name,age,sex)
28            self.birth=Date(year,mon,day) #老师有生日
29            self.courses=[] #老师有课程，可以在实例化后，往该列表中添加Course类的对象
30        def teach(self):
31            print('%s is teaching' %self.name)
32
33
34    python=Course('python','3mons',3000.0)
35    linux=Course('linux','5mons',5000.0)
36    teacher1=Teacher('lili','female',28,'博士生导师',1990,3,23)
37
38    # teacher1有两门课程
39    teacher1.courses.append(python)
40    teacher1.courses.append(linux)
41
42    # 重用Date类的功能
43    teacher1.birth.tell_birth()
44
45    # 重用Course类的功能
46    for obj in teacher1.courses:
47        obj.tell_info()

```

## 6.12、Mixins机制

Mixins机制指的是子类混合(mixin)不同类的功能，而这些类采用统一的命名规范（例如Mixin后缀），以此标识这些类只是用来**混合功能**的

```

1    class Vehicle: # 交通工具
2        pass
3
4
5    class FlyableMixin:
6        def fly(self):

```

```

7         '''
8         飞行功能相应的代码
9         '''
10        print("I am flying")
11
12
13    class CivilAircraft(FlyableMixin, vehicle): # 民航飞机
14        pass
15
16
17    class Helicopter(FlyableMixin, vehicle): # 直升飞机
18        pass
19
20
21    class Car(vehicle): # 汽车
22        pass
23
24    # ps: 采用某种规范（如命名规范）来解决具体的问题是python惯用的套路

```

可以看到，上面的CivilAircraft、Helicopter类实现了多继承，不过它继承的第一个类我们起名为FlyableMixin，而不是Flyable，这个并不影响功能，但是会告诉后来读代码的人，这个类是一个Mixin类，表示混入(mix-in)，这种命名方式就是用来明确地告诉别人（python语言惯用的手法），这个类是作为功能添加到子类中，而不是作为父类，

## 6.13、多态

- 所谓多态，定义时的类型和运行时的类型是不一样，此时就成为多态

```

1    #要想实现多态，必须有两个前提需要遵守：
2    '''
3    1.继承：多态必须发生在父类和子类之间
4    2.重写：子类重写父类的方法
5    '''
6    class Animal:
7        '''
8        基类【父类】
9        '''
10       def say(self):
11           print('我是一个动物'*10)
12   class Dark(Animal):
13       '''
14       子类【派生类】
15       '''
16       def say(self):
17           '''
18           重写父类方法
19           '''
20           print('我是一个鸭子'*10)
21   class Dog(Animal):
22       def say(self):
23           print('我是一只🐶'*10)
24   # dark=Dark()
25   # dark.say()
26
27

```

```

28 #统一的去调用
29 def commonInvoke(obj):
30     obj.say()
31 list=[Dark(),Dog()]
32 for item in list:
33     '''
34     循环调用函数
35     '''
36     commonInvoke(item)
37

```

## 6.14、类属性和实例属性

1. 类属性：就是类对象拥有的属性，它被所有类对象的实例对象所共有，类对象和实例对象可以访问
2. 实例属性：实例对象所拥有的属性，只能通过实例对象访问

```

1 class Student:
2     name = '黎明' #属于类属性，就是Student类对象所拥有
3     def __init__(self,age):
4         self.age=age #实例属性
5
6 lm=Student(18)
7 print(lm.name)
8 print(lm.age)
9 print(Student.name) #通过Student类对象访问name属性
10 # print(Student.age) #类对象不能访问实例属性
11
12 #类对象可以被类对象和实例对象共同访问使用的
13 #实例对象只能实例对象访问
14
15

```

## 6.15、类方法和静态方法

- 类方法的第一个参数是类对象cls,通过cls引用的类对象的属性和方法
- 实例对象的第一个参数是实例对象self,通过self引用的可能是类属性，也可能是实例属性，不过在存在相同名称的类属性和实例属性的情况下，实例属性的优先级更高
- 静态方法中不需要额外定义参数，因此在静态方法中引用类属性的话，必须通过类对象来引用

```

1 class People:
2     country='china'
3     @classmethod #类方法 用classmethod进行修饰
4     def get_country(cls):
5         return cls.country #访问类属性
6     @classmethod
7     def change_country(cls,data):
8         cls.country=data #修改列属性的值，在类方法中
9     @staticmethod
10    def gatData():
11        return People.country
12    @staticmethod
13    def add(x,y):

```

```

14         return x+y
15
16 print(People.gatData())
17 p=People()
18 print(p.gatData())#注意：一般情况下，我们不会通过实例对象去访问静态方法
19 print(People.add(1,2))
20 # print(People.get_country())#通过类对象去引用
21 # p=People()
22 # print('通过实例对象访问',p.get_country())
23 # People.change_country('英国')
24 # print(People.get_country())
25
26 #由于静态方法主要来存放逻辑性的代码，本身和类以及实例对象没有交互
27 #也就是说，在静态方法中，不会涉及到类中方法和属性的操作
28 #数据资源能够得到有效的充分利用
29
30
31 import time
32 class TimeTest:
33     def __init__(self,hour,minute,second):
34         self.hour=hour
35         self.minute=minute
36         self.second=second
37
38     @staticmethod
39     def showTime():
40         return time.strftime('%H:%M:%S',time.localtime())
41 print(TimeTest.showTime())
42 # t=TimeTest(2,10,11)
43 # print(t.showTime())#没必要通过实例对象访问静态方法
44

```

## 6.16、私有化属性(隐藏)

- 语法：
  - 两个下划线开头，声明该属性为私有，不能在类的外部被使用或直接访问
- 使用私有化属性的场景
  1. 把特定的一个属性隐藏起来，不想让类的外部进行直接调用
  2. 我想保护这个属性，不想让属性的值随意的改变，
  3. 保护这个属性，不想让派生类【子类】去继承
- 要想访问私有变量一般是写两个方法，一个访问，一个修改，由方法去控制访问

```

class Person:
    __age =18 #实例一个私有化属性，属性名字前面加两个下划线
    ...

```

```

1 #print(Person.__dict__)
2
3 class Person:
4     __hobby = 'dance'
5
6     def __init__(self):

```

```

7         self.__name = '李四' # 加两个下划线将此属性私有化,不能再外部直接访问,在类的
      内部可以访问
8         self.age = 30
9         print(self.__name)#print(self._Person__name)
10
11
12     def __str__(self):
13         return '{}的年龄是{}, 喜欢{}'.format(self.__name, self.age,
Person.__hobby) # 调用私有化属性
14
15     def change(self, hobby):
16         Person.__hobby = hobby
17
18
19 class Student(Person):
20     def printInfo(self):
21         # print(self.__name)#访问不了父类中的私有属性
22         print(self.age)
23
24     pass
25
26
27 x1 = Person()
28 # print(x1.name)#通过类对象,在外部访问的
29 print(x1)
30 x1.change('唱歌') # 修改私有属性的值
31 print(x1)
32 stu = Student()
33 stu.printInfo()
34 stu.change('Rap')
35 print(stu)
36 # print(x1.hobby) # 通过实例对象访问类属性
37 # print(stu.hobby) # 通过实例对象访问类属性
38 # print(Person.hobby) # 通过类对象访问类属性
39

```

- 小结:

1. 私有化的【实例】属性,不能在外部直接的访问,可以在类的内部随意的使用
2. 子类不能继承父类的私有化属性,【只能继承父类公共的属性和行为】
3. 在属性名的前面直接加\_, 就可以将其私有化

- 单下划线、双下划线、头尾双下划线三者区别

xxx前面加下划线,以单下划线开头表示的是protected类型的变量,即保护类型只能允许其本身与子类进行访问,不能使用from xxx import \*的方式导入

xxx前后两个下划线,魔术方法,一般是python自带的,开发者不要创建这类型的方法

xxx后面单下划线,避免属性名与Python关键字冲突



## 6.17、私有化方法(隐藏)

- 概述
  - 私有化方法跟私有化属性一样，有些重要的方法，不允许外部调用，防止子类意外重写，把普通的方法设置成私有化方法
  - 私有化方法一般是类内部调用，子类不能继承外部不能调用
- 语法

```
class A:  
    def __myname(self): #在方法名前面加两个下划线  
        print('小明')
```

```
1  
2 class Animal:  
3     def __eat(self):  
4         print('吃东西')  
5  
6     def run(self):  
7         self.__eat() # 在此调用私有化方法  
8         print('飞快地跑')  
9  
10  
11 class Bird(Animal):  
12     pass  
13  
14  
15 bird = Bird()  
16 bird.run()  
17
```

## 6.18、Property函数

```
1 # 属性函数 (property)  
2 class Perons:  
3     def __init__(self):  
4         self.__age = 18  
5  
6     def ger_age(self):#访问私有实例属性  
7         return self.__age  
8  
9     def set_age(self, age):#修改私有实例属性  
10        if age < 0:  
11            print('年龄太小')  
12        else:  
13            self.__age = age  
14  
15        #定义一个类属性，实现通过直接访问属性的形式去访问私有属性  
16        age=property(ger_age,set_age)  
17 p=Perons()  
18 print(p.age)  
19 p.age=-1  
20 print(p.age)  
21
```

装饰器property，可以将类中的函数“伪装成”对象的数据属性，对象在访问该特殊属性时会触发功能的执行，然后将返回值作为本次访问的结果，例如

```
1 >>> class People:
2 ...     def __init__(self,name,weight,height):
3 ...         self.name=name
4 ...         self.weight=weight
5 ...         self.height=height
6 ...     @property
7 ...     def bmi(self):
8 ...         return self.weight / (self.height**2)
9 ...
10 >>> obj=People('lili',75,1.85)
11 >>> obj.bmi #触发方法bmi的执行，将obj自动传给self，执行后返回值作为本次引用的结果
12 21.913805697589478
```

使用property有效地保证了属性访问的一致性。另外property还提供设置和删除属性的功能，如下

```
1 >>> class Foo:
2 ...     def __init__(self,val):
3 ...         self.__NAME=val #将属性隐藏起来
4 ...     @property
5 ...     def name(self):
6 ...         return self.__NAME
7 ...     @name.setter
8 ...     def name(self,value):
9 ...         if not isinstance(value,str): #在设定值之前进行类型检查
10 ...             raise TypeError('%s must be str' %value)
11 ...         self.__NAME=value #通过类型检查后,将值value存放到真实的位置self.__NAME
12 ...     @name.deleter
13 ...     def name(self):
14 ...         raise PermissionError('Can not delete')
15 ...
16 >>> f=Foo('lili')
17 >>> f.name
18 lili
19 >>> f.name='LiLi' #触发name.setter装饰器对应的函数name(f,'Egon')
20 >>> f.name=123 #触发name.setter对应的的函数name(f,123),抛出异常TypeError
21 >>> del f.name #触发name.deleter对应的函数name(f),抛出异常PermissionError
```

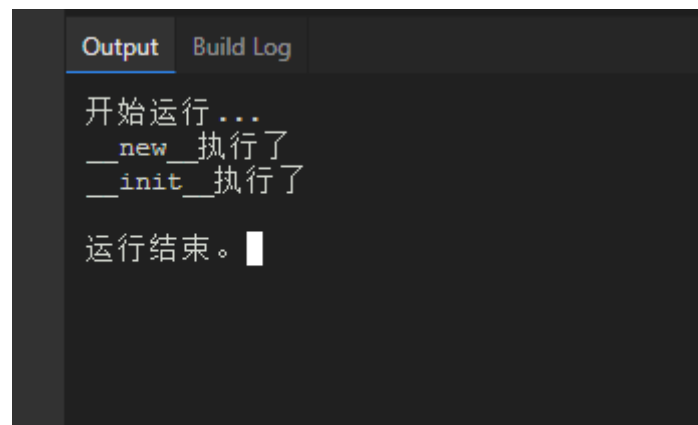
## 6.19、\_\_new\_\_ 方法

## 概述

- `__new__` 方法的作用是，创建并返回一个实例对象，如果 `__new__` 只调用了一次，就会得到一个对象。继承自 `object` 的新式类才有 `new` 这一魔法方法。
- 注意事项：
  - `__new__` 是在一个对象实例化的时候所调用的第一个方法
  - `__new__` 至少必须要有一个参数 `cls`，代表要实例化的类，此参数在实例化时由Python解释器自动提供，其他的参数是用来直接传递给 `__init__` 方法
  - `__new__` 决定是否要使用该 `__init__` 方法，因为 `__new__` 可以调用其他类的构造方法或者直接返回别的实例对象来作为本类的实例，如果 `__new__` 没有返回实例对象，则 `__init__` 不会被调用
  - 在 `__new__` 方法中，不能调用自己的 `__new__` 方法，即： `return cls.__new__(cls)`，否则会报错（`RecursionError: maximum recursion depth exceeded`：超过最大递归深度）
- `new` 方法在 `init` 方法之前执行

```
1 class A(object):
2     def __init__(self):
3         print("__init__执行了")
4
5     def __new__(cls, args, **kwargs):
6         print("__new__执行了")
7         return object.__new__(cls) # 调用父类的new方法
8
9
10 a = A() # 实例化的过程会自动调用__new__方法取创建实例
11
```

运行结果如下：



```
Output Build Log
开始运行...
__new__执行了
__init__执行了
运行结束。
```

## 6.20、单例模式

单例模式：是一种常用的软件设计模式

**目的：**确保某一个类只有一个实例存在

如果希望在某个系统中只能出现一个实例，那么单例对象就能满足要求

```
1 # 创建一个单例对象,基于__new__去实现
2
3
4 class DataBaseClass(object):
```

```

5     def __new__(cls, *args, **kwargs):
6         # cls._instance=cls.__new__(cls)不能使用自身的new方法，容易造成深度递归，应
        该调用父类的new方法
7         if not hasattr(cls, "_instance"): # 如果不存在，就开始创建
8             cls._instance = super().__new__(cls, *args, **kwargs)
9         return cls._instance
10
11
12 db1 = DataBaseClass()
13 print(id(db1))
14 db2 = DataBaseClass()
15 print(id(db2))
16

```

## 6.21、异常处理

不需要在每个可能出错的地方捕获，只要在合适的层次去捕获错误就可以

```

1 #语法格式:
2 try:
3     可能出错的代码块
4 except:
5     出错后执行的代码块
6 else:
7     没有出错的代码块
8 finally:
9     不管有没有错都执行的代码块

```

### try-except

- except在捕获错误类型的时候，主要根据具体的错误类型来捕获的
- 用一个try可以捕获多个不同类型的异常

code 1:

```

1 try:
2     print(b) #这里是要捕获的代码
3
4 except NameError: #指定错误类型为NameError
5     #捕捉到错误会在这里执行
6     print("NameError")
7

```

code 2:

```

1 try:
2     print(b) #这里是要捕获的代码
3
4 except NameError as msg:
5     #捕捉到错误会在这里执行
6     print(msg)
7

```

code 3:

```

1  try:
2
3      li=[1,2,54,78] #这里是要捕获的代码
4      print(li[10])
5      a=10/0
6  except NameError as msg:
7      #捕捉到错误会在这里执行
8      print(msg)
9
10 except IndexError as msg:
11     print(msg)
12 except ZeroDivisionError as msg:
13     print(msg)
14

```

code 4:

```

1  try:
2      a=10/0
3  except Exception as msg:
4      print(msg)
5

```

code 5:

```

1  def A(s):
2      return 10/int(s)
3  def B(s):
4      return A(s)*2
5  def main():
6      try:
7          B("0")
8      except Exception as msg:
9          print(msg)
10
11 main()
12 #不需要在每个可能出错的地方捕获，只要在合适的层次去捕获错误就可以

```

### try-except-else

```

1  try:
2      print (aa)
3  except Exception as msg:
4      print(msg)
5  else:
6      print("当try里面没有错误。else才会执行")

```

### try-except-else-finally

```

1 try:
2     print (aa)
3 except Exception as msg:
4     print(msg)
5 else:
6     print("当try里面没有错误。else才会执行")
7 finally:
8     print("不过有没有错都执行")

```

### 自定义异常

- 自定义异常，都要直接或者间接的继承 `Error` 或 `Exception` 类
- 由开发者主动抛出自定义异常，在python中使用 `raise` 关键字

```

1 class ToolsMyException(Exception):
2     def __init__(self, leng):
3         self.leng = leng
4
5     def __str__(self):
6         return "您输入的的姓名长度是"+str(self.leng)+"已经超过长度"
7
8 def name_Test():
9     name=input("请输入姓名")
10    try:
11        if len(name)>5:
12            raise ToolsMyException(len(name))
13        else:
14            print(name)
15    except ToolsMyException as result:
16        print(result)
17    finally:
18        print("执行完毕-----")
19
20 name_Test()

```

## 6.22、反射机制

[详见相关博客](#)

## 6.23、元类

[详见相关博客](#)

# 07、文件操作

文件是操作系统提供给用户/应用程序操作硬盘的一个虚拟的概念/接口

用户/应用程序可以通过文件将数据永久保存在硬盘中

用户/应用程序直接操作的是文件，对文件进行的所有的操作，都是在向操作系统发送系统调用，然后再由操作系统将其转成具体的硬盘操作

## 7.1、文件操作流程

- 打开文件

打开文件，由应用系统向操作系统发起系统调用`open()`，操作系统打开该文件，对应一块硬盘空间，并返回一个文件对象赋值给一个变量`f`

```
1  """Windows路径分隔符问题"""
2  """方案1 （推荐）"""
3  f=open(r'C:\a\b\c\d.txt',mode='rt')#r原生字符串rawstring
4  """方案2"""
5  f=open(r'C:/a/b/c/d.txt')
```

- 操作文件（读/写）

调用文件对象下的读/写方法，会被操作系统转换为读/写硬盘的操作

```
1  f.read()
```

- 关闭文件

向操作系统发起关闭文件的请求，回收系统资源

```
1  f.close()
```

### with上下文管理

```
1  #1. 执行完代码后，with会自动执行fp.close()
2  with open('a.txt',mode='rt') as fp:
3      res=fp.read()#t模式会将fp.read()读出来的结果解码成unicode
4      print(res)
5  #2. 用with打开多个文件，用逗号分割开即可
6  with open('a.txt','r') as f1,open('b.txt','r') as f2:
7      res1=f1.read()
8      res2=f2.read()
9      print(res1)
10     print(res2)
```

### 指定操作文件的字符编码

```
1  """
2  f=open(...)由操作系统打开文件，如果打开的是文本文件，会涉及到字符编码问题，如果没有为open
   指定编码，那么打开文本文件的默认编码是操作系统说了算的，
3  操作系统会用自己默认的编码去打开文件，在windos下是gbk，在Linux下是utf-8
4  """
```

## 7.2、文件的操作模式

### 控制文件读写操作的模式

```
1 """
2     r（默认的）只读
3     w: 只写
4     a: 只追加写
5 """
```

- r模式使用

```
1 # r只读模式：在文件不存在时报错，文件存在文件内指针直接跳到文件开头
2 with open('a.txt',mode='r',encoding='utf-8') as f:
3     res=f.read() # 会将文件的内容由硬盘全部读入内存，赋值给res
4
5 # 小练习：实现用户认证功能
6 inp_name=input('请输入你的名字: ').strip()
7 inp_pwd=input('请输入你的密码: ').strip()
8 with open('db.txt',mode='r',encoding='utf-8') as f:
9     for line in f:
10         # 把用户输入的名字与密码与读出内容做比对
11         u,p=line.strip('\n').split(':')
12         if inp_name == u and inp_pwd == p:
13             print('登录成功')
14             break
15     else:
16         print('账号名或者密码错误')
```

- w模式的使用

```
1 # w只写模式：在文件不存在时会创建空文档，文件存在会清空文件，文件指针跑到文件开头
2 with open('b.txt',mode='w',encoding='utf-8') as f:
3     f.write('你好\n')
4     f.write('我好\n')
5     f.write('大家好\n')
6     f.write('111\n222\n333\n')
7 #强调：
8 # 1 在文件不关闭的情况下，连续的写入，后写的内容一定跟在前写内容的后面
9 # 2 如果重新以w模式打开文件，则会清空文件内容
10
11 #案例：w模式用来创建全新的文件
12 #文件的copy工具
13 src_file=input("源文件路径》》").strip()
14 dsc_file=input("源文件路径》》").strip()
15 with open(r'{}'.format(src_file),mode='rt',encoding='utf-8')as f1,open(r'{}'.format(dsc_file),mode='wt',encoding='utf-8')as f2:
16     res=f1.read()
17     f2.write(res)
```

- a模式的使用

```
1 # a只追加写模式：在文件不存在时会创建空文档，文件存在会将文件指针直接移动到文件末尾
```



```

2  with open('c.txt',mode='a',encoding='utf-8') as f:
3      f.write('44444\n')
4      f.write('55555\n')
5  #强调 w 模式与 a 模式的异同:
6  # 1 相同点: 在打开的文件不关闭的情况下, 连续的写入, 新写的内容总会跟在前写的内容之后
7  # 2 不同点: 以 a 模式重新打开文件, 不会清空原文件内容, 会将文件指针直接移动到文件末尾, 新
   写的内容永远写在最后
8
9  # 小练习: 实现注册功能:把用户名和密码添加至数据库
10 name=input('username>>>: ').strip()
11 pwd=input('password>>>: ').strip()
12 with open('db1.txt',mode='a',encoding='utf-8') as f:
13     info='%s:%s\n' %(name,pwd)
14     f.write(info)

```

## • +模式的使用

```

1  # r+ w+ a+ :可读可写
2  #在平时工作中, 我们只单纯使用r/w/a, 要么只读, 要么只写, 一般不用可读可写的模式

```

## 控制文件读写内容的模式

```

1  大前提: tb模式均不能单独使用, 必须与r/w/a之一结合使用
2  t (默认的): 文本模式
3      1. 读写文件都是以字符串为单位的
4      2. 只能针对文本文件
5      3. 必须指定encoding参数
6  b: 二进制模式:
7      1. 读写文件都是以bytes/二进制为单位的
8      2. 可以针对所有文件
9      3. 一定不能指定encoding参数

```

## • t模式的使用

```

1  # t 模式: 如果我们指定的文件打开模式为r/w/a, 其实默认就是rt/wt/at
2  with open('a.txt',mode='rt',encoding='utf-8') as f:
3      res=f.read()
4      print(type(res)) # 输出结果为: <class 'str'>
5
6  with open('a.txt',mode='wt',encoding='utf-8') as f:
7      s='abc'
8      f.write(s) # 写入的也必须是字符串类型
9
10 #强调: t 模式只能用于操作文本文件, 无论读写, 都应该以字符串为单位, 而存取硬盘本质都是二进制的形式, 当指定 t 模式时, 内部帮我们做了编码与解码

```

## • b模式的使用

```

1  # b: 读写都是以二进制位单位
2  with open('1.mp4',mode='rb') as f:
3      data=f.read()
4      print(type(data)) # 输出结果为: <class 'bytes'>
5

```

```

6  with open('a.txt',mode='wb') as f:
7      msg="你好"
8      res=msg.encode('utf-8') # res为bytes类型
9      f.write(res) # 在b模式下写入文件的只能是bytes类型
10
11 #强调: b模式对比t模式
12 1、在操作纯文本文件方面t模式帮我们省去了编码与解码的环节, b模式则需要手动编码与解码, 所以此
    时t模式更为方便
13 2、针对非文本文件(如图片、视频、音频等)只能使用b模式
14
15 # 小练习: 编写拷贝工具
16 src_file=input('源文件路径: ').strip()
17 dst_file=input('目标文件路径: ').strip()
18 with open(r'%s' %src_file,mode='rb') as read_f,open(r'%s'
    %dst_file,mode='wb') as write_f:
19     for line in read_f:
20         # print(line)
21         write_f.write(line)

```

## 7.3、操作文件的方法

### 重点

```

1  # 读操作
2  f.read() # 读取所有内容,执行完该操作后, 文件指针会移动到文件末尾
3  f.readline() # 读取一行内容,光标移动到第二行首部
4  f.readlines() # 读取每一行内容,存放于列表中
5
6  # 强调:
7  # f.read()与f.readlines()都是将内容一次性读入内容, 如果内容过大会导致内存溢出, 若还想将
    内容全读入内存, 则必须分多次读入, 有两种实现方式:
8  # 方式一
9  with open('a.txt',mode='rt',encoding='utf-8') as f:
10     for line in f:
11         print(line) # 同一时刻只读入一行内容到内存中
12
13 # 方式二
14 with open('1.mp4',mode='rb') as f:
15     while True:
16         data=f.read(1024) # 同一时刻只读入1024个Bytes到内存中
17         if len(data) == 0:
18             break
19         print(data)
20
21 # 写操作
22 f.write('1111\n222\n') # 针对文本模式的写,需要自己写换行符
23 f.write('1111\n222\n'.encode('utf-8')) # 针对b模式的写,需要自己写换行符
24 f.writelines(['333\n','444\n']) # 文件模式
25 f.writelines([bytes('333\n',encoding='utf-8'),'444\n'.encode('utf-8')]) #b模
    式

```

### 了解

```

1 f.readable() # 文件是否可读
2 f.writable() # 文件是否可读
3 f.closed # 文件是否关闭
4 f.encoding # 如果文件打开模式为b,则没有该属性
5 f.flush() # 立刻将文件内容从内存刷到硬盘
6 f.name

```

## 7.4、主动移动文件内指针移动

```

1 #大前提:文件内指针的移动都是Bytes为单位的,唯一例外的是t模式下的read(n),n以字符为单位
2 with open('a.txt',mode='rt',encoding='utf-8') as f:
3     data=f.read(3) # 读取3个字符
4
5
6 with open('a.txt',mode='rb') as f:
7     data=f.read(3) # 读取3个Bytes
8
9
10 # 之前文件内指针的移动都是由读/写操作而被动触发的,若想读取文件某一特定位置的数据,则需要
    用f.seek方法主动控制文件内指针的移动,详细用法如下:
11 # f.seek(指针移动的字节数,模式控制):
12 # 模式控制:
13 # 0: 默认的模式,该模式代表指针移动的字节数是以文件开头为参照的
14 # 1: 该模式代表指针移动的字节数是以当前所在的位置为参照的
15 # 2: 该模式代表指针移动的字节数是以文件末尾的位置为参照的
16 # 强调:其中0模式可以在t或者b模式使用,而1跟2模式只能在b模式下用

```

- 0模式

只有0模式可以在t下使用, 1, 2必须在b模式下使用

```

1 # a.txt用utf-8编码,内容如下(abc各占1个字节,中文“你好”各占3个字节)
2 abc你好
3
4 # 0模式的使用
5 with open('a.txt',mode='rt',encoding='utf-8') as f:
6     f.seek(3,0) # 参照文件开头移动了3个字节
7     print(f.tell()) # 查看当前文件指针距离文件开头的位置,输出结果为3
8     print(f.read()) # 从第3个字节的位置读到文件末尾,输出结果为:你好
9     # 注意:由于在t模式下,会将读取的内容自动解码,所以必须保证读取的内容是一个完整中文数
    据,否则解码失败
10
11 with open('a.txt',mode='rb') as f:
12     f.seek(6,0)
13     print(f.read().decode('utf-8')) #输出结果为:好

```

- 1模式

```

1 # 1模式的使用
2 with open('a.txt',mode='rb') as f:
3     f.seek(3,1) # 从当前位置往后移动3个字节，而此时的当前位置就是文件开头
4     print(f.tell()) # 输出结果为: 3
5     f.seek(4,1) # 从当前位置往后移动4个字节，而此时的当前位置为3
6     print(f.tell()) # 输出结果为: 7

```

## • 2模式

```

1 # a.txt用utf-8编码，内容如下（abc各占1个字节，中文“你好”各占3个字节）
2 abc你好
3
4 # 2模式的使用
5 with open('a.txt',mode='rb') as f:
6     f.seek(0,2) # 参照文件末尾移动0个字节，即直接跳到文件末尾
7     print(f.tell()) # 输出结果为: 9
8     f.seek(-3,2) # 参照文件末尾往前移动了3个字节
9     print(f.read().decode('utf-8')) # 输出结果为: 好
10
11 # 小练习：实现动态查看最新一条日志的效果
12 #exe.py:
13 with open('access.log','a',encoding='utf-8')as fp:
14     fp.write('202120202020200 收款2000元\n')
15 #access_log.py:
16 import time
17 with open('access.log',mode='rb') as f:
18     f.seek(0,2)#将指针移到末尾
19     while True:
20         line=f.readline()
21         if len(line) == 0:
22             # 没有内容
23             time.sleep(0.5)
24         else:
25             print(line.decode('utf-8'),end='')
26 """每执行一次exe.py，access_log就会检测到数据，并打印出来"""

```

## 7.5文件的修改

```

1 # 文件a.txt内容如下
2 张一蛋    山东    179    49    12344234523
3 李二蛋    河北    163    57    13913453521
4 王全蛋    山西    153    62    18651433422
5
6 # 执行操作
7 with open('a.txt',mode='r+t',encoding='utf-8') as f:
8     f.seek(9)
9     f.write('<妇女主任>')
10
11 # 文件修改后的内容如下
12 张一蛋<妇女主任> 179    49    12344234523
13 李二蛋    河北    163    57    13913453521
14 王全蛋    山西    153    62    18651433422
15
16 # 强调:

```

- 17 # 1、硬盘空间是无法修改的,硬盘中数据的更新都是用新内容覆盖旧内容
- 18 # 2、内存中的数据是可以修改的

文件对应的是硬盘空间,硬盘不能修改对应着文件本质也不能修改, 那我们看到文件的内容可以修改,是如何实现的呢? 大致的思路是将硬盘中文件内容读入内存,然后在内存中修改完毕后再覆盖回硬盘

具体的实现方式分为两种:

- 方式一

```
1 # 实现思路: 将文件内容发一次性全部读入内存,然后在内存中修改完毕后再覆盖写回原文件
2 # 优点: 在文件修改过程中同一份数据只有一份
3 # 缺点: 会过多地占用内存
4 with open('db.txt',mode='rt',encoding='utf-8') as f:
5     data=f.read()
6
7 with open('db.txt',mode='wt',encoding='utf-8') as f:
8     f.write(data.replace('zhao','SB'))
```

- 方式二

```
1 # 实现思路: 以读的方式打开原文件,以写的方式打开一个临时文件,一行行读取原文件内容,修改完后写入临时文件...,删掉原文件,将临时文件重命名原文件名
2 # 优点: 不会占用过多的内存
3 # 缺点: 在文件修改过程中同一份数据存了两份
4 import os
5
6 with open('db.txt',mode='rt',encoding='utf-8') as read_f,\
7     open('.db.txt.swap',mode='wt',encoding='utf-8') as write_f:
8     for line in read_f:
9         write_f.write(line.replace('SB','kevin'))
10
11 os.remove('db.txt')
12 os.rename('.db.txt.swap','db.txt')
```

## 7.6垃圾回收机制

解释器在执行到定义变量的语法时,会申请内存空间来存放变量的值,而内存的容量是有限的,这就涉及到变量值所占用内存空间的回收问题,当一个变量值没有用了(简称垃圾)就应该将其占用的内存给回收掉,那什么样的变量值是没有用的呢?

单从逻辑层面分析,我们定义变量将变量值存起来的目的是为了以后取出来使用,而取得变量值需要通过其绑定的直接引用(如x=10,10被x直接引用)或间接引用(如l=[x],x=10,10被x直接引用,而被容器类型l间接引用),所以当变量值不再绑定任何引用时,我们就无法再访问到该变量值了,该变量值自然就是没有用的,就应该被当成一个垃圾回收。

毫无疑问,内存空间的申请与回收都是非常耗费精力的事情,而且存在很大的危险性,稍有不慎就有可能引发内存溢出问题,好在Cpython解释器提供了自动的垃圾回收机制来帮我们解决了这件事。

### 什么是GC

垃圾回收机制(简称GC)是Python解释器自带一种机制,专门用来回收不可用的变量值所占用的内存空间

### 为什么用垃圾回收机制

程序运行过程中会申请大量的内存空间，而对于一些无用的内存空间如果不及时清理的话会导致内存使用殆尽（内存溢出），导致程序崩溃，因此管理内存是一件重要且繁杂的事情，而python解释器自带的垃圾回收机制把程序员从繁杂的内存管理中解放出来。

## 垃圾回收机制原理分析

Python的GC模块主要运用了“引用计数”（reference counting）来跟踪和回收垃圾。在引用计数的基础上，还可以通过“标记-清除”（mark and sweep）解决容器对象可能产生的循环引用的问题，并且通过“分代回收”（generation collection）以空间换取时间的方式来进一步提高垃圾回收的效率。

### （一）引用计数

引用计数就是：变量值被变量名关联的次数

如：age=18

变量值18被关联了一个变量名age，称之为引用计数为1

引用计数增加：

age=18（此时，变量值18的引用计数为1）

m=age（把age的内存地址给了m，此时，m,age都关联了18，所以变量值18的引用计数为2）

引用计数减少：

age=10（名字age先与值18解除关联，再与3建立了关联，变量值18的引用计数为1）

del m（del的意思是解除变量名x与变量值18的关联关系，此时，变量18的引用计数为0）

值18的引用计数一旦变为0，其占用的内存地址就应该被解释器的垃圾回收机制回收

```
1 a=12 #直接引用
2 b=a #间接引用
```

```
1 """循环引用"""
2 l1=[111,]
3 l2=[222,]
4 l1.append(l2)#l1=[值111的内存地址, l2列表的内存地址]
5 l2.append(l1)#l2=[值222的内存地址, l1列表的内存地址]
6
7 print(id(l1[1]))
8 print(id(l2))
9
10 print(id(l2[1]))
11 print(id(l1))
12
13 print(l2)
14 print(l1[1])
15
16 """结果如下"""
17 2006853601024
18 2006853601024
```

```
19
20 2006561499520
21 2006561499520
22
23 [222, [111, [...]]]
24 [222, [111, [...]]]
```

[垃圾回收机制](#)

## 08、正则表达式

学习正则表达式操作字符串

re模块是用C语言写的没匹配速度非常快

其中compile函数根据一个模式字符串和可选的标志参数生成一个正则表达式对象，该对象拥有一系列方法用于正则表大会匹配和替换，re模块也提供了与这下方法功能完全一致的函数，这些函数适用一个模式字符串做为他们的第一个参数

### re.math方法

- re.math 尝试**从字符串起始位置匹配**，返回match对象，，否则返回None，适用group()获取匹配成功的字符串
  - 语法：re.match(pattern,string,flags)

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串
flags	标志位，用于控制正则表达式的匹配方式:如:是否匹配大小写，多行匹配

```
1 import re
2 str='python is the best language in the world'
3 result= re.match('P',str)
4 print(type(result))#<class 're.Match'>
5 print(result.group())
```

### 标志位

- 如果使用多个标志位，使用|分割，如：re.I|re.M

修饰符	描述
re.I	适匹配对大小写不敏感
re.L	做本地化识别匹配
re.M	多行匹配，影响^ 和\$
re.S	使.匹配包括换行在内的所有字符
re.U	根据Unicode字符集解析字符，这个标志影响\w, \W ,\b, \B
re.X	该标识符通过给予你更灵活的格式以便于你将正则表达式写得更易于理解。

```

1 import re
2 strData='Python is the best language in the world\
3 gslfjgldsjsjls'
4 #result= re.match('p',strData,re.I|re.M)#第三个参数 忽略大小写
5 #print(type(result))#<class 're.Match'>
6 #print(result.group())
7 res=re.match('(.*)is(.*)',strData,re.I)
8 print(res.group(1))
9 print(res.group(2))
10

```

## 常用匹配规则

符号	匹配规则
.	匹配任意1个字符除了换行符
[abc]	匹配abc中任意一个
\d	匹配一个数字0-9
\D	匹配非数字
\s	匹配空白 即空格 tab键
\S	匹配非空格
\w	匹配单词字符 即a-z A-Z 0-9 _
\W	匹配非单词字符

## 匹配字符数量

符号	匹配规则
*	匹配前一个字符出现0次或者无限次，即可有可无
+	匹配前一个字符出现1次或者无限次，即至少有1次
\?	匹配前一个字符出现1次或者0次，即要么有1次要么没有
{m}	匹配前一个字符出现m次
{m,}	匹配前一个字符至少出现m次
{m,n}	匹配前一个字符出现从m次到n次

### 8.1、限定匹配数量规则

```

1 import re
2
3 # * 匹配前一个字符出现0次或者无限次
4 res=re.match('[a-z][a-z]*','MyPython',re.I)
5 print(res.group())
6
7 # + 匹配前一个字符1次或者无限次 至少一次

```



```

8  res=re.match('[a-zA-Z]+[\w]*','mynAMEDCeisz848s_')
9  print(res.group())
10
11  # ? 匹配前一个字符0次或者1次
12  res=re.match('[a-zA-Z]+[\d]?','mkohjhjgu8jg8')
13  print(res.group())
14
15  # {min,max} 匹配前一个从min到max次 min max必须是非负整数
16  #{count}精确匹配次数 {count,}没有限制
17  res=re.match('\d{4,}','46145')
18  if res:
19      print('匹配成功{}'.format(res.group()))
20
21  #匹配邮箱 格式: xxxxxx@163.com
22  res=re.match('[a-zA-Z0-9]{6,11}@163.com','318129549@163.com')
23  print(res.group())
24
25

```

## 8.2、原生字符串

```

1  # path="D:\\1_zhao_File\\1_MarkDown\\MarkDown学习使用篇"
2  # print(path )
3  import re
4
5
6  #原生字符串 r
7  print(re.match(r'c:\\a.text','c:\\a.text').group())
8
9
10 #匹配开头结尾
11 #^ 匹配字符串开头
12 #$ 匹配字符串结尾
13 # res=re.match('^p.*','python is language')
14 res=re.match('^p[\w]{5}','python is language')
15 print(res.group())
16 res=re.match('[\w]{5,15}@[ \w]{2,5}.com$','318129549@qq.com')
17 print(res.group())
18

```

## 8.3、分组匹配

```

1  # | 匹配左右任意一个表达式 从左往右
2  import re
3
4  res=re.match('[\w]*|100','100')
5  print(res.group())
6
7  # (ab)分组匹配 将括号中字符作为一个分组
8  res=re.match('([0-9]*)-(\d*)','123456-464651561')
9  print(res.group())
10 print(res.group(1))
11 print(res.group(2))
12

```

```

13 # \num 的使用
14 # htmlTag='<html><h1>Python核心编程</h1></html>'
15 # res1=re.match(r'<(.)>(.)>(.)</\2></\1>',htmlTag)
16 # print(res1.group(1))
17
18
19 # 分组 别名的使用 (?P<名字>)
20 data='<div><h1>www.baidu.com</h1></div>'
21 res=re.match(r'<(?P<div>\w*)><(?P<h1>\w*)>(?P<data>.*</\w*></\w*>',data)
22
23 print(res.group())
24

```

## 8.4、编译函数compile

```

1 # re.compile 方法
2 '''
3 compile将正则表达式模式编译成一个正则表达式对象
4 reg=re.compile(pattern)
5 result=reg.match(string)
6 等效于result=re.match(pattern,string)
7 使用re.compile和保持所产生的正则表达式对象重用效率更高
8 '''
9 import re
10
11 #compile 可以把字符串编译成字节码
12 #优点: 在使用正则表达式进行match时, python会将字符串转为正则表达式对象
13 # 而如果使用compile, 只需要转换一次即可, 以后在使用模式对象的话无需重复转换
14
15 data='1364'
16 pattern=re.compile('.*')
17 #使用pattern对象
18 res=pattern.match(data)
19 print(res.group())
20
21
22 #re.search方法
23 #search在全文中匹配一次, 匹配到就返回
24 data='我爱我伟大的祖国, I love China,China is a great country'
25 rees=re.search('China',data)
26 print(rees)
27 print(rees.span())
28 print(rees.group())
29 # print(data[21])
30
31 #re.findall方法 匹配所有, 返回一个列表,
32
33 data='华为牛逼是华人的骄傲'
34 # res =re.findall('华.',data)
35 # print(res)
36 pattern=re.compile('华.')
37 res=pattern.findall(data)
38 print(res)
39
40

```

```

41 # re.sub方法 实现目标搜索和替换
42 data1='Python是很受欢迎的编程语言'
43 pattern='[a-zA-Z]+' #字符集范围 +代表 前导字符模式出现1从以上
44 res=re.sub(pattern,'C#',data1)
45 resn=re.subn(pattern,'C#',data1)
46 print(res)
47 print(resn)
48 #re.subn 完成目标的搜索和替换 还返回被替换的数量，以元组的形式返回
49
50 #re.split 是新分割字符串
51 data='百度，腾讯，阿里，华为，360，字节跳动'
52 print(re.split(',',data))
53

```

## 8.5贪婪模式和非贪婪模式

```

1 '''
2 python 中默认是贪婪的，总是贪婪的匹配尽可能多的字符，非贪婪相反，总是尝试匹配尽可能少的字符
3 在 " * ? + {m,n}"后面加上 ? 使贪婪变成非贪婪
4
5 '''
6 #贪婪
7 import re
8 res=re.match('[\d]{6,9}','111222333')
9 print(res.group())
10
11
12 #非贪婪
13 res=re.match('[\d]{6,9}?','111222333')
14 print(res.group())
15
16
17 content='asdfsdbdsabsd'
18 # pattern=re.compile('a.*b')# 贪婪
19 pattern=re.compile('a.*?b')#非贪婪
20 res=pattern.search(content)
21 print(res.group())
22 #0710-49
23
24 ``\xxxxxxxxx23 1'''2python 中默认是贪婪的，总是贪婪的匹配尽可能多的字符，非贪婪相反，
总是尝试匹配尽可能少的字符3在 " * ? + {m,n}"后面加上 ? 使贪婪变成非贪婪45'''6#贪婪
7import re8res=re.match('[\d]{6,9}','111222333')9print(res.group())101112#
非贪婪13res=re.match('[\d]
{6,9}?','111222333')14print(res.group())151617content='asdfsdbdsabsd'18#
pattern=re.compile('a.*b')# 贪婪19pattern=re.compile('a.*?b')#非贪婪
20res=pattern.search(content)21print(res.group())22#0710-4923python

```

25

```
``xxxxxxxxx24 1''2python 中默认是贪婪的，总是贪婪的匹配尽可能多的字符，非贪婪相反，
总是尝试匹配尽可能少的字符3在 " * ? + {m,n}"后面加上 ? 使贪婪变成非贪婪45''6#贪婪
7import re8res=re.match('[\d]{6,9}','111222333')9print(res.group())101112#
非贪婪13res=re.match('[\d]
{6,9}?','111222333')14print(res.group())151617content='asdfsdbdsabsd'18#
pattern=re.compile('a.*b')# 贪婪19pattern=re.compile('a.*?b')#非贪婪
20res=pattern.search(content)21print(res.group())22#0710-
492324``xxxxxxxxx23 1''2python 中默认是贪婪的，总是贪婪的匹配尽可能多的字符，非贪
婪相反，总是尝试匹配尽可能少的字符3在 " * ? + {m,n}"后面加上 ? 使贪婪变成非贪婪
45''6#贪婪7import re8res=re.match('[\d]
{6,9}','111222333')9print(res.group())101112#非贪婪13res=re.match('[\d]
{6,9}?','111222333')14print(res.group())151617content='asdfsdbdsabsd'18#
pattern=re.compile('a.*b')# 贪婪19pattern=re.compile('a.*?b')#非贪婪
20res=pattern.search(content)21print(res.group())22#0710-4923pythonpython
```

```

` ``xxxxxxxxxx26 1'''2python 中默认是贪婪的，总是贪婪的匹配尽可能多的字符，非贪婪相反，
总是尝试匹配尽可能少的字符3在 " * ? + {m,n}"后面加上 ? 使贪婪变成非贪婪45'''6#贪婪
7import re8res=re.match('[\d]{6,9}','111222333')9print(res.group())101112#
非贪婪13res=re.match('[\d]
{6,9}?','111222333')14print(res.group())151617content='asdfbsdbdsabsd'18#
pattern=re.compile('a.*b')# 贪婪19pattern=re.compile('a.*?b')#非贪婪
20res=pattern.search(content)21print(res.group())22#0710-
492324` ``xxxxxxxxxx23 1'''2python 中默认是贪婪的，总是贪婪的匹配尽可能多的字符，非贪
婪相反，总是尝试匹配尽可能少的字符3在 " * ? + {m,n}"后面加上 ? 使贪婪变成非贪婪
45'''6#贪婪7import re8res=re.match('[\d]
{6,9}','111222333')9print(res.group())101112#非贪婪13res=re.match('[\d]
{6,9}?','111222333')14print(res.group())151617content='asdfbsdbdsabsd'18#
pattern=re.compile('a.*b')# 贪婪19pattern=re.compile('a.*?b')#非贪婪
20res=pattern.search(content)21print(res.group())22#0710-
4923python25` ``xxxxxxxxxx24 1'''2python 中默认是贪婪的，总是贪婪的匹配尽可能多的字
符，非贪婪相反，总是尝试匹配尽可能少的字符3在 " * ? + {m,n}"后面加上 ? 使贪婪变成非贪
婪45'''6#贪婪7import re8res=re.match('[\d]
{6,9}','111222333')9print(res.group())101112#非贪婪13res=re.match('[\d]
{6,9}?','111222333')14print(res.group())151617content='asdfbsdbdsabsd'18#
pattern=re.compile('a.*b')# 贪婪19pattern=re.compile('a.*?b')#非贪婪
20res=pattern.search(content)21print(res.group())22#0710-
492324` ``xxxxxxxxxx23 1'''2python 中默认是贪婪的，总是贪婪的匹配尽可能多的字符，非贪
婪相反，总是尝试匹配尽可能少的字符3在 " * ? + {m,n}"后面加上 ? 使贪婪变成非贪婪
45'''6#贪婪7import re8res=re.match('[\d]
{6,9}','111222333')9print(res.group())101112#非贪婪13res=re.match('[\d]
{6,9}?','111222333')14print(res.group())151617content='asdfbsdbdsabsd'18#
pattern=re.compile('a.*b')# 贪婪19pattern=re.compile('a.*?b')#非贪婪
20res=pattern.search(content)21print(res.group())22#0710-
4923pythonpython26` ``xxxxxxxxxx25 1'''2python 中默认是贪婪的，总是贪婪的匹配尽可能
多的字符，非贪婪相反，总是尝试匹配尽可能少的字符3在 " * ? + {m,n}"后面加上 ? 使贪婪变
成非贪婪45'''6#贪婪7import re8res=re.match('[\d]
{6,9}','111222333')9print(res.group())101112#非贪婪13res=re.match('[\d]
{6,9}?','111222333')14print(res.group())151617content='asdfbsdbdsabsd'18#
pattern=re.compile('a.*b')# 贪婪19pattern=re.compile('a.*?b')#非贪婪
20res=pattern.search(content)21print(res.group())22#0710-
492324` ``xxxxxxxxxx23 1'''2python 中默认是贪婪的，总是贪婪的匹配尽可能多的字符，非贪
婪相反，总是尝试匹配尽可能少的字符3在 " * ? + {m,n}"后面加上 ? 使贪婪变成非贪婪
45'''6#贪婪7import re8res=re.match('[\d]
{6,9}','111222333')9print(res.group())101112#非贪婪13res=re.match('[\d]
{6,9}?','111222333')14print(res.group())151617content='asdfbsdbdsabsd'18#
pattern=re.compile('a.*b')# 贪婪19pattern=re.compile('a.*?b')#非贪婪
20res=pattern.search(content)21print(res.group())22#0710-
4923python25` ``xxxxxxxxxx24 1'''2python 中默认是贪婪的，总是贪婪的匹配尽可能多的字
符，非贪婪相反，总是尝试匹配尽可能少的字符3在 " * ? + {m,n}"后面加上 ? 使贪婪变成非贪
婪45'''6#贪婪7import re8res=re.match('[\d]
{6,9}','111222333')9print(res.group())101112#非贪婪13res=re.match('[\d]
{6,9}?','111222333')14print(res.group())151617content='asdfbsdbdsabsd'18#
pattern=re.compile('a.*b')# 贪婪19pattern=re.compile('a.*?b')#非贪婪
20res=pattern.search(content)21print(res.group())22#0710-
492324` ``xxxxxxxxxx23 1'''2python 中默认是贪婪的，总是贪婪的匹配尽可能多的字符，非贪
婪相反，总是尝试匹配尽可能少的字符3在 " * ? + {m,n}"后面加上 ? 使贪婪变成非贪婪
45'''6#贪婪7import re8res=re.match('[\d]
{6,9}','111222333')9print(res.group())101112#非贪婪13res=re.match('[\d]
{6,9}?','111222333')14print(res.group())151617content='asdfbsdbdsabsd'18#
pattern=re.compile('a.*b')# 贪婪19pattern=re.compile('a.*?b')#非贪婪

```

```
20res=pattern.search(content)21print(res.group())22#0710-  
4923pythonpythonpythonpython
```