# Python简介

1. 程序员：

程序设计人员。

1. 程序：

一组计算机能识别和执行的指令，是实现某种需求的软件。

1. 操作系统：

管理和控制计算机软件与硬件资源的程序；

隔离不同硬件的差异，使开发程序简单化。

例如，Windows，Linux，Unix。

1. 硬件：

主板--计算机的主要电路系统。

CPU --主要负责执行程序指令，处理数据。

硬盘--持久化存储数据的记忆设备，容量大，速度慢。

内存--临时存储数据的记忆设备，容量小，速度快。

IO设备—键盘、鼠标、显示器。

## Python定义



是一个**免费、开源、跨平台、动态、面向对象**的编程语言。

## Python程序的执行方式

### 交互式

在命令行输入指令，回车即可得到结果。

1. 打开终端
2. **进入交互式：python3**
3. **编写代码：print(“hello world”)**
4. **离开交互式：exit()**

### 文件式

将指令编写到.py文件，可以重复运行程序。

1. 编写文件。
2. 打开终端
3. **进入程序所在目录：cd 目录**
4. **执行程序：python3 文件名**

## Linux常用命令

文件式操作

ls #在终端显示当前文件夹中的内容

Ctrl+l #清除终端显示的内容

gedit hello.py #用Linux默认文本编辑器打开文件hello.py

print('hello world') #在文本编辑器中编写python代码

Ctrl+s #保存

pwd 查看工作目录路径

/home/tarena

/ 根 相当于window的计算机

/home 保存所有用户的目录

/home/tarena tarena用户的目录

cd 改变工作目录

cd 路径

/home/tarena 绝对路径 从根开始找目录 一直找到想要的为止

.. 相对路径 相对于当前目录的上级目录

. 相对路径 指代当前的目录

/home/tarena/work 对于tarena来说 相当于./work

/home 对于tarena来说是上一级目录 相当于../

tarena@tedu:~$ pwd

/home/tarena

tarena@tedu:~$ cd ./work

tarena@tedu:~/work$ pwd

/home/tarena/work

tarena@tedu:~/work$ cd ..

tarena@tedu:~$ pwd

/home/tarena

在Linux中创建目录和文件

**mkdir 创建目录**

mkdir 目录名

**touch 创建文件**

touch 文件名

终端中的快捷键

Tab 代码补全 如果目录中有对应代码的内容 可以快速补全

/home/tarena/a + tab

aid1907/m + tab

mounth01

Ctrl + l 清屏（将当前行移到终端最上方）

Ctrl + - 缩小终端字体

Ctrl + +(Shift =)放大终端字体

## 执行过程



计算机只能识别机器码(1010)，不能识别源代码(python)。

1. 由源代码转变成机器码的过程分成两类：编译和解释。
2. 编译：在程序运行之前，通过编译器将源代码变成机器码，例如：C语言。

-- 优点：运行速度快

--缺点：开发效率低，不能跨平台。

1. 解释：在程序运行之时，通过解释器对程序逐行翻译，然后执行。例如Javascript

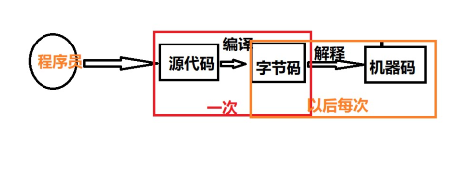
-- 优点：开发效率高，可以跨平台；

-- 缺点：运行速度慢。

1. python是解释型语言，但为了提高运行速度，使用了一种编译的方法。编译之后得到pyc文件，存储了字节码（特定于Python的表现形式，不是机器码）。

源代码 --编译--> 字节码--解释--> 机器码

|————1次———|



## 解释器类型

1. CPython（C语言开发)
2. Jython (java开发)
3. IronPython (.net开发)

# 数据基本运算

## pycharm常用快捷键

1. **移动到本行开头：home键**
2. **移动到本行末尾：end键盘**
3. **注释代码：ctrl + /**
4. **复制行：ctrl +d**
5. **选择列：鼠标左键 + alt**
6. **移动行：shift + alt + 上下箭头**
7. **智能提示：Ctrl + Space**

## 注释

给人看的，通常是对代码的描述信息。

1. 单行注释：以#号开头。
2. 多行注释：三引号开头，三引号结尾。

## 函数

表示一个功能，函数定义者是提供功能的人，函数调用者是使用功能的人。

例如：

1. print(数据)作用：将括号中的内容显示在控制台中
2. 变量 = input(“需要显示的内容”) 作用：将用户输入的内容赋值给变量

## 变量

1. 定义：关联一个对象的标识符。
2. 命名：必须是字母或下划线开头，后跟字母、数字、下划线。

不能使用关键字(蓝色)，否则发生语法错误：SyntaxError: invalid syntax。

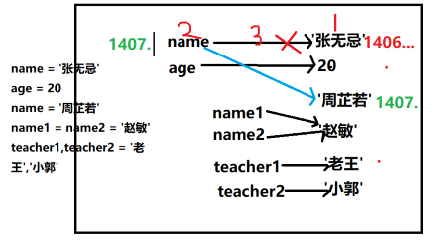
1. 建议命名：字母小写，多个单词以下划线隔开。

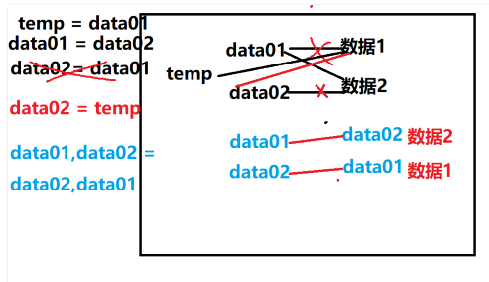
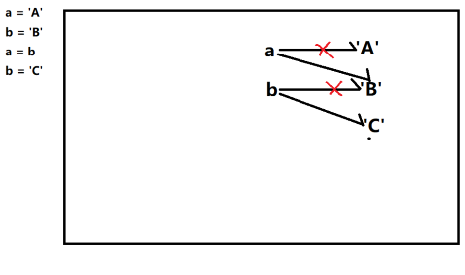
class\_name

1. 赋值：创建一个变量或改变一个变量关联的数据。
2. 语法：变量名 = 数据

变量名1 = 变量名2 =数据

变量名1, 变量名2, = 数据1, 数据2

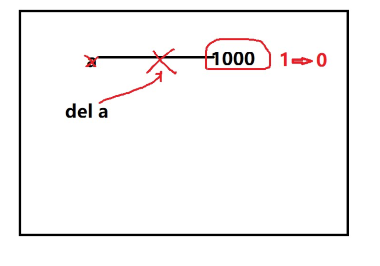




## del 语句

1. 语法:

del 变量名1, 变量名2



1. 作用：

用于删除变量,同时解除与对象的关联.如果可能则释放对象。

1. 自动化内存管理的引用计数：

每个对象记录被变量绑定(引用)的数量,当为0时被销毁。

## 核心数据类型

1. 在python中变量没有类型，但关联的对象有类型。
2. 通过type函数可查看。

### 空值对象 None

1. 表示不存在的特殊对象。
2. 作用：占位和解除与对象的关联。

### 整形int

1. 表示整数，包含正数、负数、0。

如： -5, 100, 0

1. 字面值：

十进制：5

二进制：0b开头，后跟1或者1

八进制：0o开头，后跟0~7

十六进制：0x开头，后跟0~9,A~F,a~f

1. 小整数对象池：CPython 中整数 -5 至 256,永远存在小整数对象池中,不会被释放并可重复使用。

### 浮点型float

1. 表示小数，包含正数、负数，0.0)。
2. 字面值：

小数：1.0 2.5

科学计数法：e/E (正负号) 指数

1.23e-2 (等同于0.0123)

1.23456e5(等同于123456.0)

### 字符串str

是用来记录文本信息(文字信息)。

字面值：双引号

### 复数complex

由实部和虚部组成的数字。

虚部是以j或J结尾。

字面值： 1j 1+1j 1-1j

### 布尔bool

用来表示真和假的类型

True 表示真(条件满足或成立)，本质是1

False 表示假(条件不满足或不成立)，本质是0

*#任意数据类型转换为布尔类型时  
#如果有值就为True 没有值就为False  
#字符串中包含空白字符 True*print(bool(**' '**))  
print(bool(**''**)) *#空字符串 False*print(bool(0))*# False*print(bool(0.0))*# False*print(bool(**None**))*# False*

## 数据类型转换

1. 转换为整形: int(数据)
2. 转换为浮点型:float(数据)
3. 转换为字符串:str(数据)
4. 转换为布尔:bool(数据)

结果为False：bool(0) bool(0.0) bool(None)

1. 混合类型自动升级：

1 + 2.14 返回的结果是 3.14

1 + 3.0 返回结果是: 4.0

## 运算符

### 算术运算符

+ 加法

- 减法

\* 乘法

/ 除法：结果为浮点数

// 地板除：除的结果去掉小数部分

% 求余

\*\* 幂运算

优先级从高到低：()

\*\*

\* / % //

+ -

### 增强运算符

y += x 等同于 y = y + x

y -= x 等同于 y = y - x

y \*= x 等同于 y = y \* x

y /= x 等同于 y = y / x

y //= x 等同于 y = y // x

y %= x 等同于 y = y % x

y \*\*= x 等同于 y = y \*\* x

### 比较运算符

< 小于

<= 小于等于

> 大于

>= 大于等于

== 等于

!= 不等于

返回布尔类型的值

比较运算的数学表示方式:0 <= x <= 100

### 逻辑运算符

#### 与and

表示并且的关系，一假俱假。

示例:

True and True # True

True and False # False

False and True # False

False and False # False

#### 或or

表示或者的关系，一真俱真

示例:

True or True # True

True or False # True

False or True # True

False or False # False

#### 非 not

表示取反

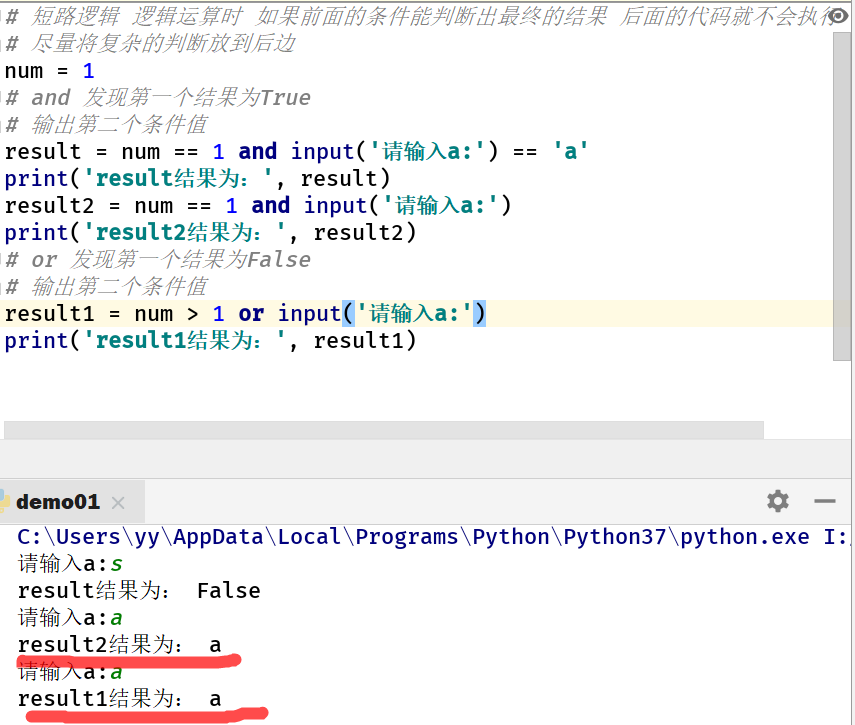
例如：

not True # 返回False

not False # 返回True

#### 短路运算

一但结果确定，后面的语句将不再执行。



### 身份运算符

语法:

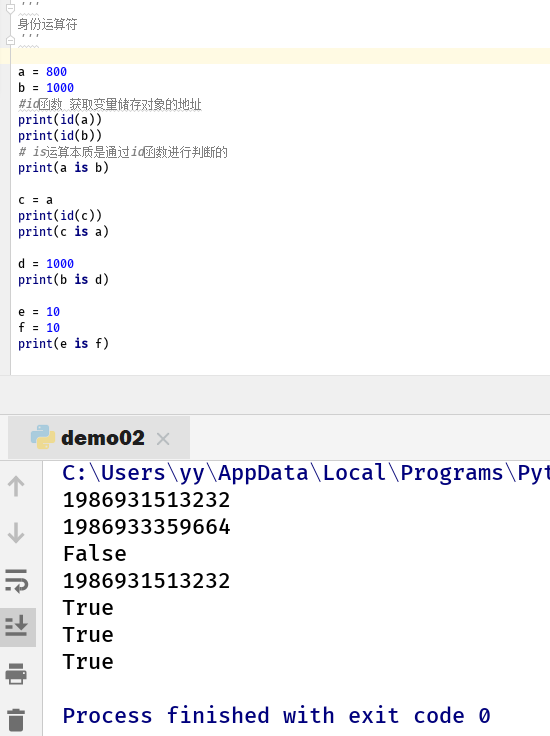
x is y

x is not y

作用：

is用于判断两个对象是否是同一个对象,是时返回True,否则返回False。

is not 的作用与is相反



### 优先级

高到低：

算数运算符

比较运算符

快捷运算符

身份运算符

逻辑运算符

# 语句

## 行

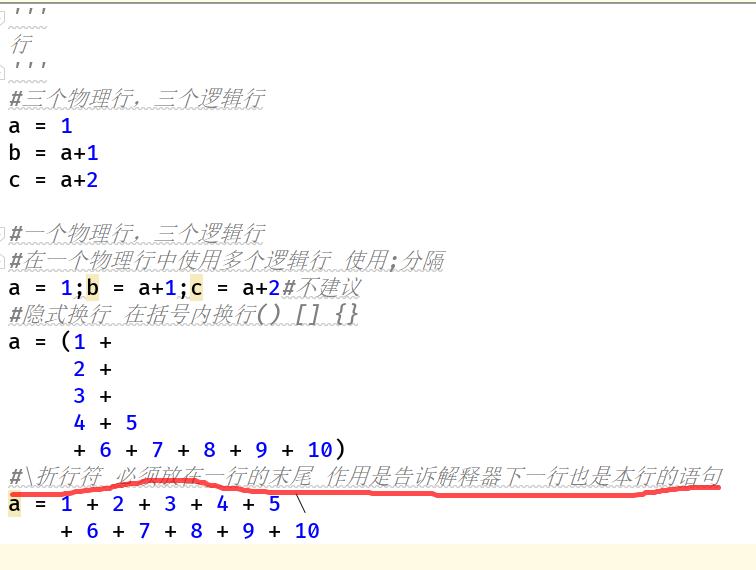
1. 物理行：程序员编写代码的行。
2. 逻辑行：python解释器需要执行的指令。
3. 建议一个逻辑行在一个物理行上。
4. 如果一个物理行中使用多个逻辑行，需要使用分号；隔开。

5. 如果逻辑行过长，可以使用隐式换行或显式换行。

隐式换行：所有括号的内容换行,称为隐式换行

括号包括: () [] {} 三种

显式换行：通过折行符 \ (反斜杠)换行，必须放在一行的末尾，目的是告诉解释器,下一行也是本行的语句。



## pass 语句

通常用来填充语法空白。

## 选择语句

### Ifelif else 语句

1. 作用:

让程序根据条件选择性的执行语句。

1. 语法:

if 条件1:

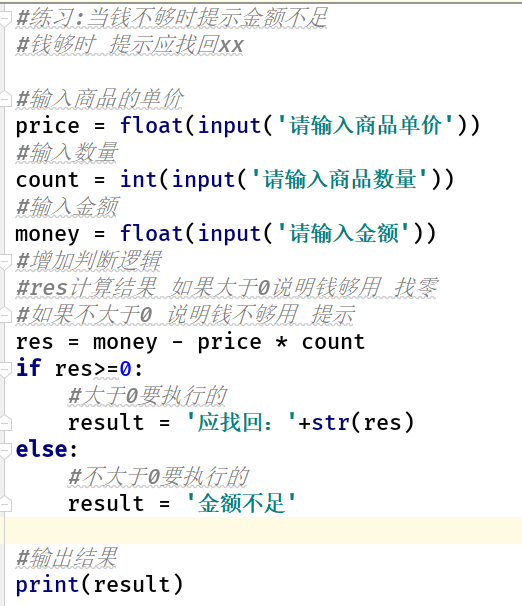
语句块1

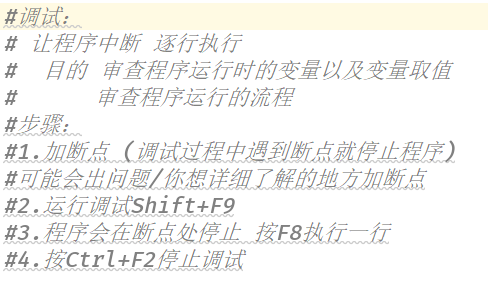
elif 条件2:

语句块2

else:

语句块3





1. 说明:

elif 子句可以有0个或多个。

else 子句可以有0个或1个，且只能放在if语句的最后。

### if 语句的真值表达式

if 100:

print("真值")

等同于

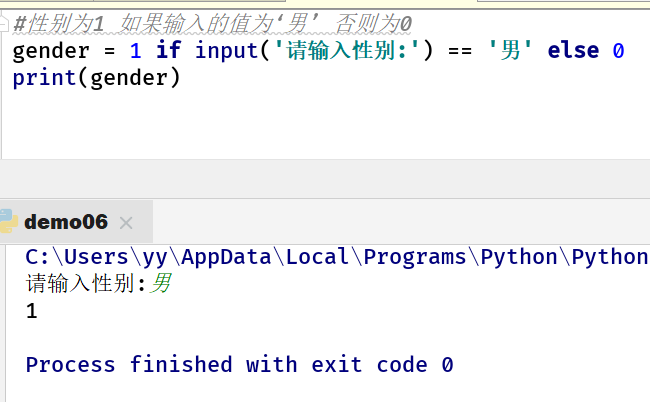
if bool(100):

print("真值")

### 条件表达式

语法：变量 = 结果1 if 条件 else 结果2

作用：根据条件(True/False) 来决定返回结果1还是结果2。



## 循环语句

### while语句

1. 作用:

可以让一段代码满足条件，重复执行。

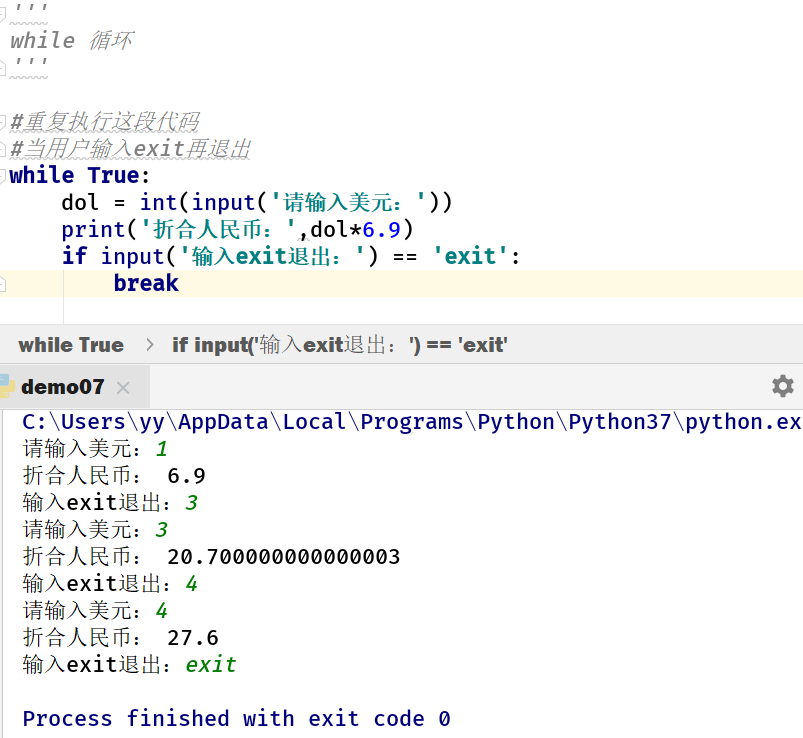
1. 语法:

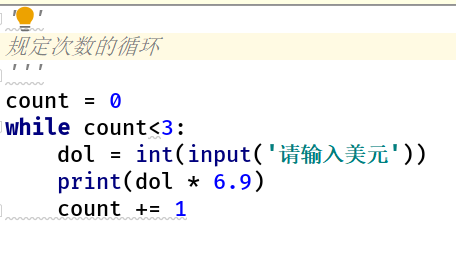
while 条件:

满足条件执行的语句

else:

不满足条件执行的语句





1. 说明:

else子句可以省略。

在循环体内用break终止循环时,else子句不执行。

### for 语句

1. 作用:

用来遍历可迭代对象的数据元素。

可迭代对象是指能依次获取数据元素的对象，例如：容器类型。

1. 语法:

for 变量列表 in 可迭代对象:

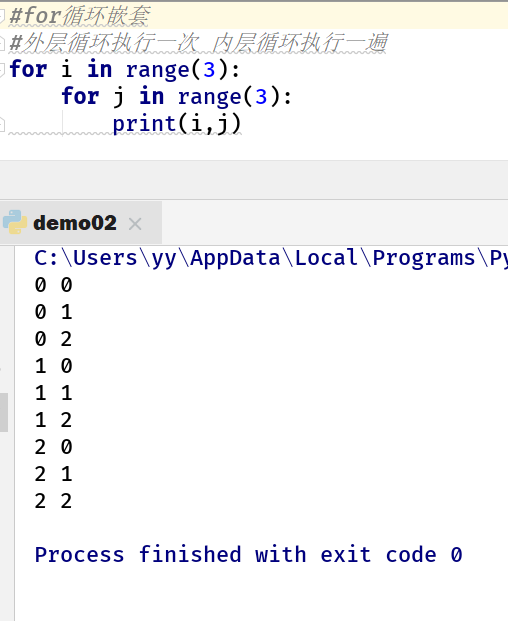
语句块1

else:

语句块2



### for 循环嵌套语句



1. 说明:

else子句可以省略。

在循环体内用break终止循环时,else子句不执行。

### range 函数

1. 作用:

用来创建一个生成一系列整数的可迭代对象(也叫整数序列生成器)。

1. 语法:

range(开始点，结束点，*间隔*)

1. 说明:

函数返回的可迭代对象可以用for取出其中的元素

返回的数字不包含结束点

开始点默认为0

间隔默认值为1

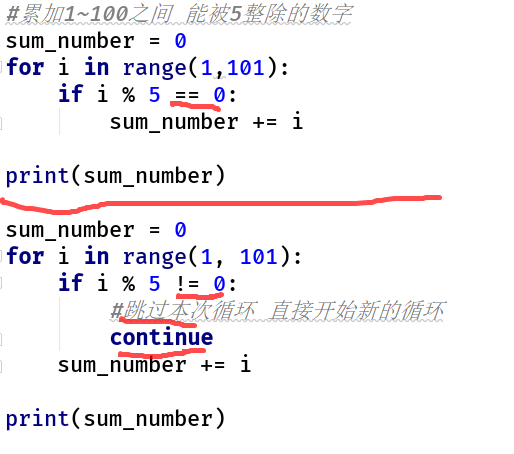
## 跳转语句

### break 语句

1. 跳出循环体，后面的代码不再执行。
2. 可以让while语句的else部分不执行。

### continue 语句

跳过本次，继续下次循环。



# 容器类型

## 通用操作

### 数学运算符

1. +：用于拼接两个容器
2. +=：用原容器与右侧容器拼接,并重新绑定变量
3. \*：重复生成容器元素
4. \*=：用原容器生成重复元素, 并重新绑定变量
5. <<= >>= == !=：依次比较两个容器中元素,一但不同则返回比较结果。

### 成员运算符

1. 语法：

数据 in 序列

数据 not in 序列

1. 作用：

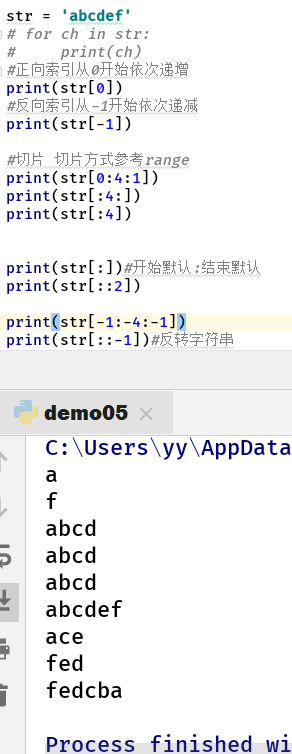
如果在指定的序列中找到值，返回bool类型。

### 索引index

1. 作用：访问容器元素
2. 语法：容器[整数]
3. 说明：

正向索引从0开始，第二个索引为1，最后一个为len(s)-1。

反向索引从-1开始,-1代表最后一个,-2代表倒数第二个,以此类推,第一个是-len(s)。



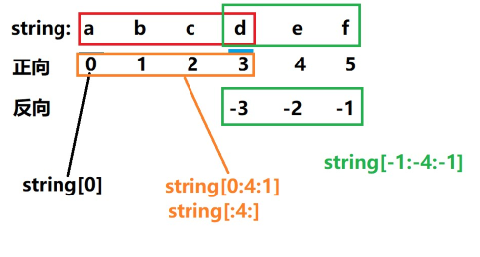
### 切片slice

1. 作用：

从容器中取出相应的元素重新组成一个容器。

1. 语法：

容器[(开始索引):(结束索引)(:(步长))]



1. 说明：

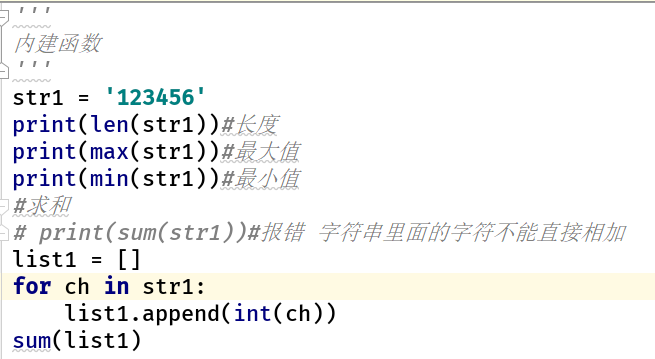
小括号()括起的部分代表可省略

结束索引不包含该位置元素

步长是切片每次获取完当前元素后移动的偏移量

### 内建函数

1. len(x) 返回序列的长度
2. max(x) 返回序列的最大值元素
3. min(x) 返回序列的最小值元素
4. sum(x) 返回序列中所有元素的和(元素必须是数值类型)



## 字符串 str

### 定义

由一系列字符组成的不可变序列容器，存储的是字符的编码值。

### 编码

1. 字节byte：计算机最小存储单位，等于8 位bit.
2. 字符：单个的数字，文字与符号。
3. 字符集(码表)：存储字符与二进制序列的对应关系。
4. 编码：将字符转换为对应的二进制序列的过程。
5. 解码：将二进制序列转换为对应的字符的过程。
6. 编码方式：

--ASCII编码：包含英文、数字等字符，每个字符1个字节。

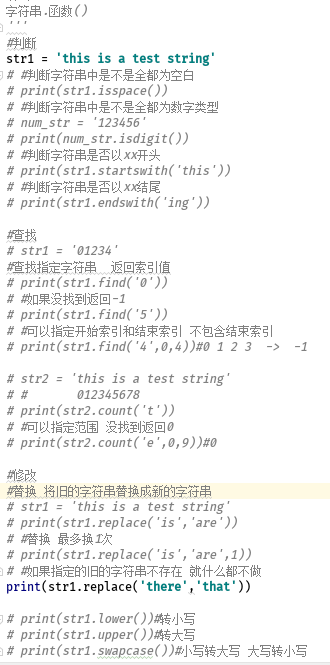
--GBK编码：兼容ASCII编码，包含21003个中文；英文1个字节，汉字2个字节。

--Unicode字符集：国际统一编码，旧字符集每个字符2字节，新字符集4字节。

-- UTF-8编码：Unicode的存储与传输方式，英文1字节，中文3字节。

#### 相关函数

1. ord(字符串):返回该字符串的Unicode码。
2. chr(整数):返回该整数对应的字符串。



### 字面值

#### 单引和双引号的区别

1. 单引号内的双引号不算结束符
2. 双引号内的单引号不算结束符

#### 三引号

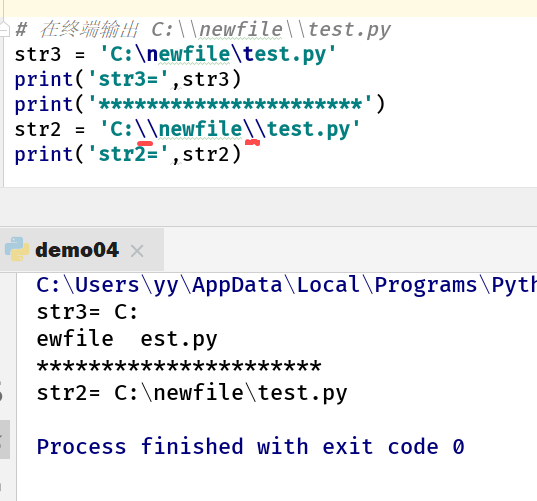
#### 作用

1. 换行会自动转换为换行符\n
2. 三引号内可以包含单引号和双引号
3. 作为文档字符串

#### 转义字符

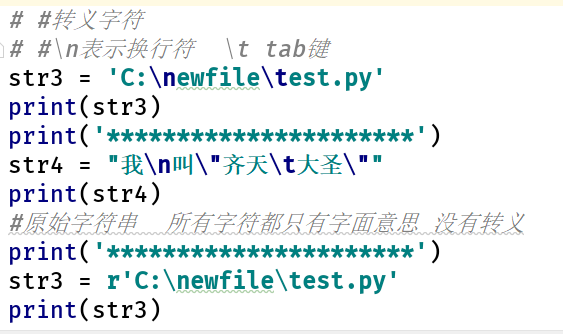
1. 改变字符的原始含义。

\’ \” \””” \n \\ \t \0空字符



1. 原始字符串：取消转义。

a = r”C:\newfile\test.py”



#### 字符串格式化

1. 定义：

生成一定格式的字符串。

1. 语法：

字符串%(变量)

"我的名字是%s,年龄是%s" % (name, age)

1. 类型码：

%s 字符串 %d整数 %f 浮点数

## 列表list

### 定义

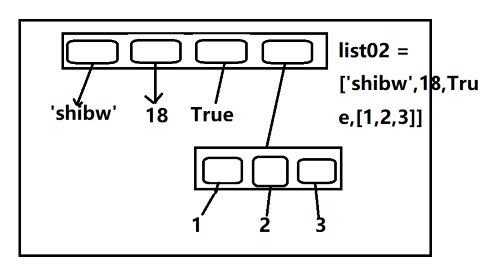
由一系列变量组成的可变序列容器。

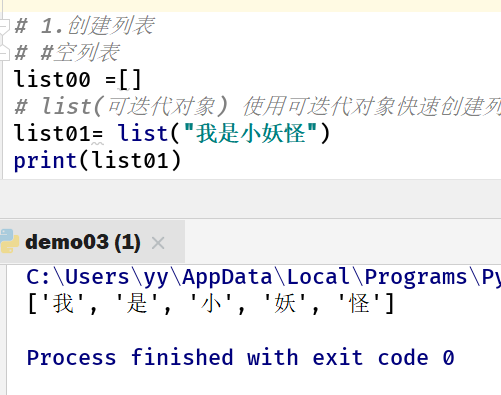
### 基础操作

1. 创建列表：

列表名 = []

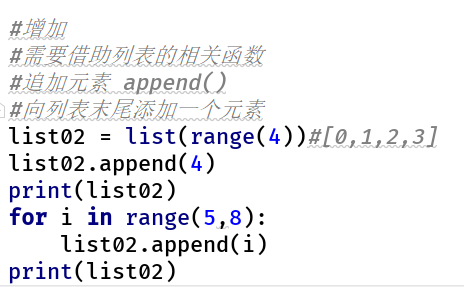
列表名 =list(可迭代对象)



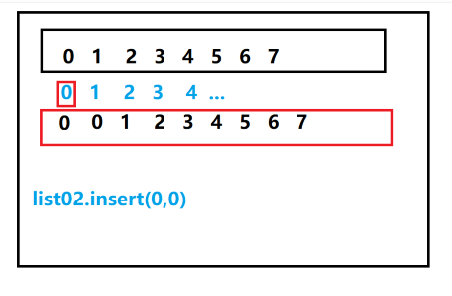


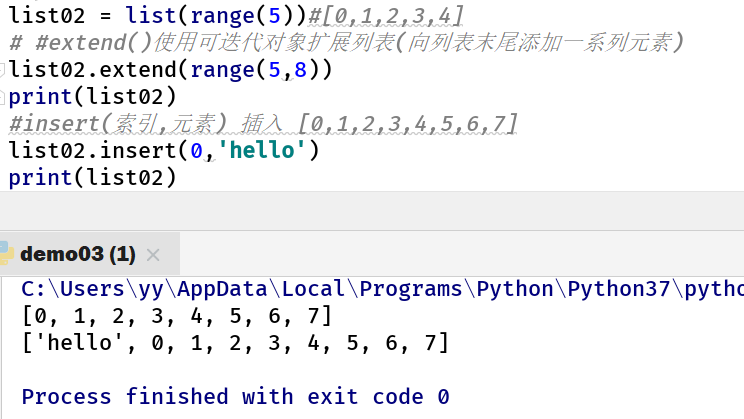
1. 添加元素：

**列表名.append(元素)**



**列表.insert(索引，元素)**





1. 定位元素：

索引、切片list01 = [0,1,2,3,4,5]  
print(list01[0:5])*#[0,1,2,3,4]*print(list01[::-1])*#[5,4,3,2,1,0]*print(list01[-1])

*# 修改*list01[-1] = 5.0  
print(list01)  
print(list01[-2:])*#倒数第二个 到最后一个  
  
# 获取列表的所有元素*list01 = [0,1,2,3,4]  
**for** item **in** list01:  
 print(item)

1. 遍历列表：

正向：

for变量名 in列表名:

变量名就是元素

反向：

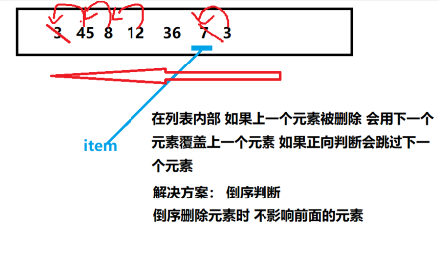
for 索引名 inrange(len(列表名)-1,-1,-1):

列表名[索引名]就是元素

1. 删除元素：

列表名.remove(元素)

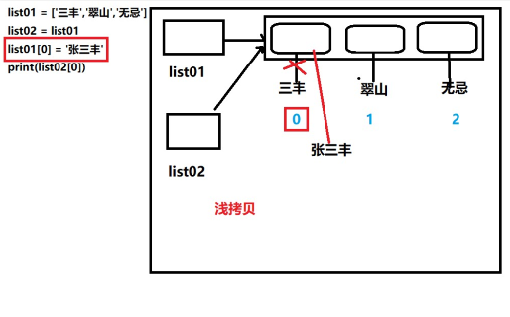
del列表名[索引或切片]

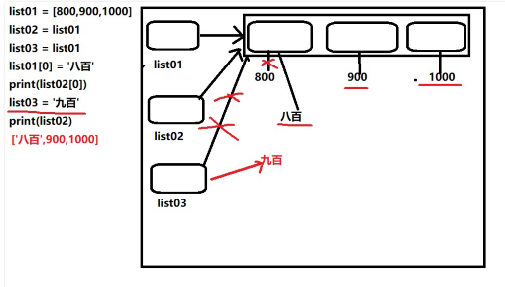


*#根据元素删除*list01 = [0,1,2,3,4]  
list01.remove(4)  
print(list01)  
*#根据索引***del** list01[3]  
print(list01)

### 深拷贝和浅拷贝

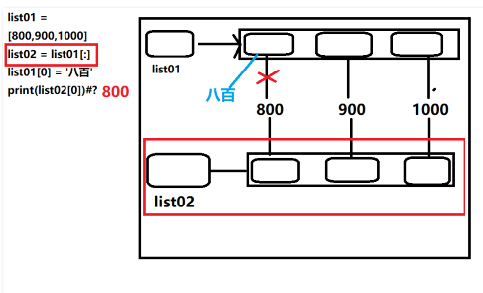
浅拷贝：复制过程中,只复制一层变量,不会复制深层变量绑定的对象的复制过程。



list01 = [800,900,1000]  
*#把list01中储存的地址赋值给list02*  


list02 = list01  
list03 = list01  
list01[0] = **'八百'**print(list02[0])*#?'八百'*list03 = **'九百'**

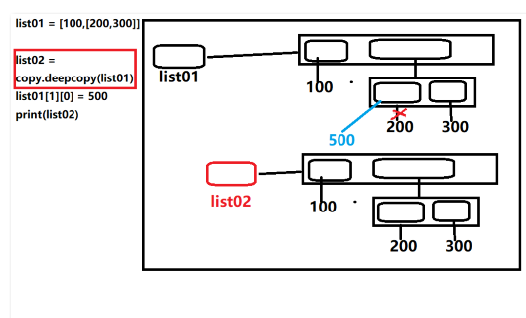
print(list03)*#'九百'*print(list02)*#['八百', 900, 1000]*

*# 切片赋值是新建一个列表，原表改变不影响切片*

list01 = [800,900,1000]  
list02 = list01[:]*#切片本质是新建一个列表 [800,900,1000]*list01[0] = **'八百'**print(list02[0])*# 800*

*# 修改列表不新建表，把list01中储存的地址赋值给list02*list01 = [800,900,1000]  
list02 = list01  
list01[1:2] = [**'a'**,**'b'**]  
print(list02)*#[800,'a','b',1000]*

深拷贝：复制整个依赖的变量。



*#准备拷贝工具***import** copy  
  
list01 = [100,[200,300]]  
  
*#浅拷贝  
#把list01中储存的地址赋值给list02*list02 = list01.copy()  
list01[1][0] = 500  
print(list02[1][0])*#500  
  
#深拷贝 划清界限 拷贝前的对象和拷贝后的对象互不影响  
#注意 深拷贝可能会占用大量内存*list02 = copy.deepcopy(list01)  
list01[1][0] = 500  
print(list02)*#[100, [200, 300]]*

### 列表VS字符串

1. 列表和字符串都是序列,元素之间有先后顺序关系。
2. 字符串是不可变的序列,列表是可变的序列。
3. 字符串中每个元素只能存储字符,而列表可以存储任意类型。
4. 列表和字符串都是可迭代对象。
5. 函数：

将多个字符串拼接为一个。

result = "连接符".join(列表)

*'''  
list --> str  
'''*list01 = [**'a'**,**'b'**,**'c'**]  
str01 = **'+'**.join(list01)  
print(str01)  
print(type(str01))

将一个字符串拆分为多个。

列表 = “a-b-c-d”.split(“分隔符”)

*'''  
str --> list  
'''*str01 = **'2019/08/06'**list\_result = str01.split(**'/'**)  
print(list\_result) *#['2019', '08', '06']*

*#将一句英文的单词反转  
#How are you --> you are How  
#将字符串按照空格拆分成列表  
#将列表反转  
#将列表中的元素拼接为字符串*message = **'How are you'**temp\_list = message.split(**' '**)  
message = **" "**.join(temp\_list[::-1])  
print(message)

### 斐波那契数列

*#斐波那契数列*fibs = [0,1]  
*#[0,1,1]  
#[0,1,1,2]  
# #[0,1,1,2,3,5,8,11...]  
# first = fibs[0]  
# second = fibs[1]  
# for i in range(13):  
# result = first + second  
# fibs.append(result)  
# #当结果添加到列表后 将前两个数替换  
# first = second  
# second = result  
# print(fibs)***for** i **in** range(13):  
 fibs.append(fibs[-1]+fibs[-2])  
print(fibs)

### 列表推导式

1. 定义：

使用简易方法，将可迭代对象转换为列表。

1. 语法：

变量 = [表达式 for 变量 in 可迭代对象]

*'''  
列表推导式  
快速的将可迭代对象变成列表  
'''*list01 = [10,1,15,5,6,7]  
list03 = [item+1 **for** item **in** list01]  
print(list03)

变量 = [表达式 for 变量 in 可迭代对象 if 条件]

list01 = [10,1,15,5,6,7]  
list05 = [item+1 **for** item **in** list01 **if** item >= 10]  
print(list05) *#[11, 16]*

1. 说明:

如果if真值表达式的布尔值为False,则可迭代对象生成的数据将被丢弃。

### 列表推导式嵌套

1. 语法：

变量 = [表达式 for 变量1 in 可迭代对象1 for 变量2 in可迭代对象2]

1. 传统写法：

result = []

for r in ["a", "b", "c"]:

for c in ["A", "B", "C"]:

result.append(r + c)

1. 推导式写法：

result = [r + c for r in list01 for c in list02]

*#列表推导式嵌套*list01 = [1,2,3]  
list02 = [4,5,6]  
*# list03 = []  
# # 将两个列表中所有的值分别相加 将结果保存到list03  
# for item1 in list01:  
# for item2 in list02:  
# list03.append(item1+item2)  
#*list03 = [item1+item2 **for** item1 **in** list01 **for** item2 **in** list02]  
print(list03)*#[5, 6, 7, 6, 7, 8, 7, 8, 9]*

## 元组 tuple

### 定义，不可变，不能赋值、删元素

1. 由一系列变量组成的不可变序列容器。
2. 不可变是指一但创建，不可以再添加/删除/修改元素。

### 基础操作

1. 创建空元组：

元组名 =()

元组名 =tuple()

*#创建元组*tuple01 = ()*#空元组*tuple0 = tuple()  
print(tuple01)*#()*print(tuple0)*#()*

### 元组只有一个元素，必须加逗号

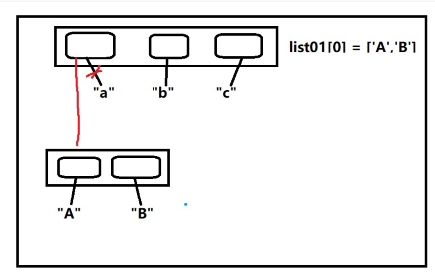
tuple01 = (**'a'**)*#不是元组*print(type(tuple01))*#<class 'str'>*tuple02 = (**'a'**,)*#如果元组中只有一个元素 必须加逗号*print(type(tuple02))*#<class 'tuple'>*name,age = **'shibw'**,18*#('shibw',18)*

1. 创建非空元组：

**元组名 = (20,)**

tuple02 = (0,1,**'nihao'**,**True**)  
print(tuple02)*#(0, 1, 'nihao', True)*

*#预留空间的存储机制(列表) --> 按需分配的存储机制(元组)*

list01 = [**'a'**,**'b'**]  
tuple02 = tuple(list01)  
print(tuple02) # ('a', 'b')  
list02 = list(tuple02)  
print(list02) # ['a', 'b']

元组名 = (1, 2, 3)

元组名 = 100,200,300

元组名 =tuple(可迭代对象)

1. 获取元素：

索引、切片

tuple01 = (**'a'**,**'b'**,**'c'**,**'d'**)  
print(tuple01[2]) #c

print(tuple01[1:3]) #('b', 'c')

1. 遍历元组：

正向：

for变量名 in列表名:

变量名就是元素

**for** item **in** tuple01:  
 print(item)

反向：

for 索引名 inrange(len(列表名)-1,-1,-1):

元组名[索引名]就是元素

**for** i **in** range(len(tuple01)-1,-1,-1):  
 print(tuple01[i])

### 元组无法改变 元组中的列表可以改变

tuple01 = ([1,2],3,4)  
print(tuple01[0][1])*#2*tuple01[0][1] = 2.0*#元组无法改变 元组中的列表可以改变*print(tuple01)*#([1, 2.0], 3, 4)*

### 作用

1. 元组与列表都可以存储一系列变量，由于列表会预留内存空间，所以可以增加元素。
2. 元组会按需分配内存，所以如果变量数量固定，建议使用元组，因为占用空间更小。
3. 应用：

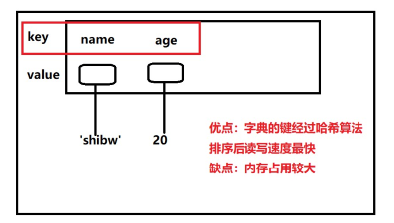
变量交换的本质就是创建元组：x, y = y, x

格式化字符串的本质就是创建元祖："姓名:%s, 年龄:%d" % ("tarena", 15)

## 字典 dict

### 定义

1. 由一系列键值对组成的可变散列容器。
2. 散列：对键进行哈希运算，确定在内存中的存储位置，每条数据存储无先后顺序。
3. 键必须惟一且不可变(字符串/数字/元组)，值没有限制。



### 基础操作

1. 创建字典：

字典名 ={键1：值1，键2：值2}

字典名 =dict (可迭代对象)

1. 添加/修改元素：

语法:

字典名[键]=数据

说明:

键不存在，创建记录。

键存在，修改映射关系。

1. 获取元素：

变量 = 字典名[键]# 没有键则错误

1. 遍历字典：

for键名 in字典名:

字典名[键名]

for键名,值名 in字典名.items():

语句

1. 删除元素：

del字典名[键]

*'''  
字典的基础操作  
'''  
  
#创建字典*dict01 = {}*#空字典*dict01 = dict()  
print(dict01)  
  
*#"shibw":20  
# 键 ：值 键值对*dict02 = {**"shibw"**:1311794174}  
*#字典中的数据以键值对的形式保存  
#多个键值对用逗号分隔*dict02 = {**'shibw'**:20,**'wwc'**:30}  
  
*# 通过函数创建字典*dict02 = dict([[**'name'**,**'shibw'**]])  
print(dict02)*#{'name': 'shibw'}*dict02 = dict([[**'name'**,**'shibw'**],(**'age'**,20)])  
print(dict02)*#{'name': 'shibw', 'age': 20}  
  
# #添加元素*dict01 = {**'name'**:**'shibw'**,**'age'**:20}  
*#如果字典中没有对应的键存在  
#可以直接对不存在的键赋值 向字典中添加该键值对*dict01[**'address'**] = **'beijing'**print(dict01)  
*#  
# #修改  
# #如果对已存在的键赋值 相当于修改原本键的值  
# dict01['address'] = 'qiaowan'  
# print(dict01)  
# #  
# #删除  
# del dict01['address']  
# print(dict01)  
  
#查找字典中的元素*dict01 = {**'name'**:**'shibw'**,**'age'**:20}  
print(dict01[**'name'**])  
*# print(dict01['address'])#KeyError***if 'name' in** dict01:  
 print(dict01[**'name'**])  
**if 'address' in** dict01:  
 print(dict01[**'address'**])  
  
*#字典的常用方法  
#使用get 获取字典指定键的值*print(dict01.get(**'name'**))  
*# 如果键不存在 可以指定返回默认值*print(dict01.get(**'address'**,**'beijing'**))  
*# 同get 但是键不存在时 会添加键值对*print(dict01.setdefault(**'address'**,**'beijing'**))  
print(dict01)  
dict01.popitem()  
print(dict01)  
  
dict01 = {**'name'**:**'shibw'**,**'age'**:20}  
dict02 = {**'name'**:**'laowang'**,**'address'**:**'北京'**}  
*#使用02更新字典01 如果02中的键在01中不存在 添加  
#如果02的键在01中存在 使用02的值*dict01.update(dict02)  
print(dict01)  
  
*#获取字典所有元素***for** key **in** dict01:  
 print(key)*#获取的是字典中所有的键* print(dict01[key])  
  
*#获取键值对(键，值)***for** item **in** dict01.items():  
 print(item)  
**for** key,value **in** dict01.items():  
 print(key)  
 print(value)  
  
*# 获取字典的所有的值***for** value **in** dict01.values():  
 print(value)  
  
dict01 = {1:**"1.0"**,2:**"2.0"**}  
print(dict01)  
dict01 = {(1,2,3):3,(1,2,3,4):4}  
print(dict01)  
*#可变类型无法通过哈希算法运算  
#所以字典的键都是不可变类型  
# dict01 = {[1,2,3]:3}#TypeError*print(dict01)

### 字典推导式

1. 定义：

使用简易方法，将可迭代对象转换为字典。

1. 语法:

{键:值 for 变量 in 可迭代对象}

{键:值 for 变量 in 可迭代对象 if 条件}

*'''  
字典推导式  
'''  
#需求 range(5) 作为key 值为key的平方*dict01 = {}  
**for** i **in** range(5):  
 dict01[i] = i\*\*2  
print(dict01)*#{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}  
#字典推导式*dict01 = {i:i\*\*2 **for** i **in** range(5)}  
print(dict01)*#{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}  
#需求 range(10)大于5的 作为key 值为key的平方*dict02 = {}  
**for** i **in** range(5):  
 **if** i > 5:  
 dict02[i] = i\*\*2  
print(dict02)*#{}*dict02 = {i:i\*\*2 **for** i **in** range(5) **if** i > 5}  
print(dict02)*#{}*

### 字典 VS 列表

1. 都是可变容器。
2. 获取元素方式不同,列表用索引,字典用键。
3. 字典的插入,删除,修改的速度快于列表。
4. 列表的存储是有序的,字典的存储是无序的。

## 集合 set

### 定义

1. 由一系列不重复的不可变类型变量组成的可变散列容器。
2. 相当于只有键没有值的字典(键则是集合的数据)。



### 基础操作

1. 创建空集合：

集合名 =set()

集合名 =set(可迭代对象)

1. 创建具有默认值集合：

集合名 ={1, 2, 3}

集合名 =set(可迭代对象)

1. 添加元素：

集合名.add(元素)

1. 删除元素：

集合名.discard(元素)

*'''  
集合的基础操作  
'''  
  
#创建空集合*set01 = set()*#set()空集合*print(set01)  
*#创建有两个元素的集合*set02 = {**'a'**,**'b'**}  
print(set02)  
*#集合中不包含重复元素*set02 = set(**'abcabc'**)  
print(set02)*#{'a','b','c'}  
  
#添加元素  
#{'a','b','c'}*set02.add(**'lvze'**)  
print(set02)  
set02.add(**'mly'**)  
print(set02)  
*#add 将一个对象放入集合  
#列表是可变类型 不能放入集合（字符串、元组、数字）  
# set02.add([0,1,2])#报错  
  
# {'c', 'a', 'mly', 'lvze', 'b'}  
#删除元素*set02.remove(**'a'**)  
print(set02)  
*#如果要删除的元素不存在 报错  
# set02.remove('x')#KeyError  
  
#获取所有***for** item **in** set02:  
 print(item)

### 运算

1. 交集&：返回共同元素。

s1 = {1, 2, 3}

s2 = {2, 3, 4}

s3 = s1 & s2 # {2, 3}

1. 并集：返回不重复元素

s1 = {1, 2, 3}

s2 = {2, 3, 4}

s3 = s1 | s2 # {1, 2, 3, 4}

1. 补集-：返回只属于其中之一的元素

s1 = {1, 2, 3}

s2 = {2, 3, 4}

s1 - s2 # {1} 属于s1但不属于s2

补集^：返回不同的的元素

s1 = {1, 2, 3}

s2 = {2, 3, 4}

s3 = s1 ^ s2 # {1, 4} 等同于(s1-s2 | s2-s1)

1. 子集<：判断一个集合的所有元素是否完全在另一个集合中
2. 超集>：判断一个集合是否具有另一个集合的所有元素

s1 = {1, 2, 3}

s2 = {2, 3}

s2 < s1 # True

s1 > s2 # True

1. 相同或不同== !=：判断集合中的所有元素是否和另一个集合相同。

s1 = {1, 2, 3}

s2 = {3, 2, 1}

s1 == s2 # True

s1 != s2 # False

子集或相同,超集或相同<= >=

*'''  
集合的运算  
交集 并集 补集  
'''  
  
#交集 &  
# s1 = {1,2,3}  
# s2 = {2,3,4}  
# #求s1和s2的交集  
# #交集：两个集合中共同的元素  
# s3 = s1 & s2  
# print(s3)#{2, 3}  
  
# shift+ \ --> |  
#并集 '|'  
#返回两个集合中所有不重复的元素  
# s1 = {1,2,3}  
# s2 = {2,3,4}  
# s3 = s1 | s2 #{1,2,3,4}  
# print(s3)  
  
#补集 -  
# s1 = {1,2,3}  
# s2 = {2,3,4}  
# s3 = s1 - s2#属于s1但不属于s2  
# print(s3)#{1}  
#  
# #补集 ^  
# s1 = {1,2,3}  
# s2 = {2,3,4}  
# s3 = s1 ^ s2#等同于(s1-s2|s2-s1)  
# print(s3)  
  
  
# s1 = {1,2,3}  
# s2 = {1,2}  
# #子集  
# #< 判断一个集合所有元素是否完全在另一个集合中  
# #如果s1中包含了s2的全部元素 我们就把s2称为s1的子集  
# print(s2<s1)#True  
# #超集(super)  
# #> 判断一个集合是否包含另一个集合的所有元素  
# #如果s1中包含了s2的全部元素 我们就把s1称为s2的超集  
# print(s1>s2)#True  
  
#相同或不相同  
# 判断集合中所有元素是否和另一个集合元素相同  
# ==  
# s1 = {1,2,3}  
# s2 = {3,2,1}  
# print(s1 == s2)#True  
# print(s1 != s2)#False  
  
#集合推导式  
# 将0~9的数字放入集合  
# set\_result = set()  
# for i in range(10):  
# set\_result.add(i)  
# print(set\_result)  
# set\_result = {i for i in range(10)}  
# print(set\_result)  
#  
# # set\_result = set()  
# # for i in range(10):  
# # if i % 2:  
# # set\_result.add(i)  
# set\_result = {i for i in range(10) if i % 2}  
# print(set\_result)  
  
  
  
#固定集合  
#主要作用：与其他容器对象转换*s1 = frozenset([**'d'**,**'a'**,**'b'**])  
print(s1)  
  
s2 = frozenset([**'a'**,**'b'**,**'c'**])  
  
*# print(s1<s2)*print(s1 & s2)  
print(s1 | s2)  
print(s1 ^ s2)  
  
*#固定集合不能添加、修改、删除  
#其余运算同set*

### 集合推导式

1. 定义：

使用简易方法，将可迭代对象转换为集合。

1. 语法:

{表达式 for 变量 in 可迭代对象}

{表达式 for 变量 in 可迭代对象 if 条件}

## 固定集合 frozenset

### 定义

不可变的集合。

#### 作用

固定集合可以作为字典的键,还可以作为集合的值。

#### 基础操作

创建固定集合：frozenset(可迭代对象)

#### 运算

等同于set

## 容器对象对比

***#字符串 str()*** *#只能装字符 不可变序列  
#列表 list()  
#任意数据类型 可变序列  
#元组 tuple()  
#任意数据类型 不可变序列  
  
#序列的通用操作  
# + 拼接两个序列  
# \* 生成重复的序列  
# += \*=  
# 索引 / 切片  
  
#字典 dict()  
#键必须是不可变类型 值可以为任意对象  
#可变散列对象  
#集合 set()  
#只能包含不可变类型 可变散列对象  
#固定集合 frozenset() 不可变散列对象  
  
# None  
# int float bool complex str list tuple dict set****#可变 list dict set  
#不可变 数字 str tuple*** *# and 一假俱假  
# or 一真俱真  
#短路逻辑  
# not****# 语句*** *# if  
# while  
# for*

# 函数 function

## pycharm相关设置

1. “代码自动完成”时间延时设置

File -> Settings-> Editor -> General -> Code Completion-> Autopopup in (ms):0

1. 快捷键：

Ctrl + P 参数信息（在方法中调用参数）

Ctrl + Q 快速查看文档

Ctrl + Alt + M 提取方法

## 定义

1. 用于封装一个特定的功能，表示一个功能或者行为。
2. 函数是可以重复执行的语句块, 可以重复调用。

## 作用

提高代码的可重用性和可维护性（代码层次结构更清晰）。

## 定义函数

1. 语法：

def 函数名(形式参数):

  函数体

1. 说明：

def 关键字：全称是define，意为”定义”。

函数名：对函数体中语句的描述，规则与变量名相同。

形式参数：方法定义者要求调用者提供的信息。

函数体：完成该功能的语句。

1. 函数的第一行语句建议使用文档字符串描述函数的功能与参数。

**def** attack(count):  
 *'''  
 攻击敌人* **:param** *count:int 攻击次数* **:return***:  
 '''* **for** i **in** range(count):  
 print(**'左勾拳'**)  
 print(**'右勾拳'**)  
 print(**'黑虎掏心'**)  
 print(**'天马流星拳'**)  
 print(**'临门一脚'**)  
 *#... ...  
  
#用功能  
#函数名保存的是对应功能的代码的地址*print(attack)  
*#当函数定义完成后 程序不会马上执行函数  
#当函数调用者用过函数名()的方式调用函数时  
#函数中的代码才会被执行*attack(1)  
attack(2)

## 调用函数

1. 语法：函数名(实际参数)
2. 说明：根据形参传递内容。

## 返回值

1. 定义：

方法定义者告诉调用者的结果。

1. 语法：

return 数据

1. 说明：

return后没有语句，相当于返回 None。

函数体没有return，相当于返回None。

*#所有函数都有返回值  
#函数通过return语句将值返回给函数调用者  
#返回的位置就是函数调用的位置  
#如果没有写return语句 默认返回None***def** myadd(num1,num2):  
 *# result = num1+num2  
 # return result  
 #返回结果 退出函数* **return** num1+num2  
 *#return语句后的代码不会执行* print(num1+num2)  
 *#要不要写return语句取决于用户是否需要再次处理函数的结果  
 #如果需要再次处理就必须通过return返回结果  
  
# print(myadd(10,20))*result = myadd(10,20)  
*#输出 结果是30*print(**'结果是%s'** % result )

## 可变／不可变类型在传参时的区别

1. 不可变类型参数有:

数值型(整数，浮点数,复数)

布尔值bool

None 空值

字符串str

元组tuple

固定集合frozenset

1. 可变类型参数有:

列表 list

字典 dict

集合 set

1. 传参说明：

不可变类型的数据传参时，函数内部不会改变原数据的值。

可变类型的数据传参时，函数内部可以改变原数据。

## 函数参数

### 实参传递方式argument

#### 位置传参

定义：实参与形参的位置依次对应。

##### 序列传参

定义：实参用\*将序列拆解后与形参的位置依次对应。

#### 关键字传参

定义：实参根据形参的名字进行对应。

##### 字典关键字传参

1. 定义：实参用\*\*将字典拆解后与形参的名字进行对应。
2. 作用：配合形参的缺省参数，可以使调用者随意传参。

*'''  
函数传参  
实参传递方式  
'''***def** fun01(a,b,c):*#形参 a b c* print(a)  
 print(b)  
 print(c)  
  
*#位置实参 实参是根据位置与形参对应的  
#如果实参位置发生改变 会影响函数结果  
# fun01(10,20,30)  
# fun01(30,10,20)  
  
#序列实参 用\*将序列中的元素拆开然后与形参依次对应  
#序列 字符串 列表 元组  
# list01 = [10,20,30]  
# fun01(\*list01)  
# str01 = 'abcd'  
# fun01(\*str01)#报错  
  
#关键字实参  
#实参的值与形参的名称对应  
# fun01(a=10,b=20,c=30)  
#使用关键字实参 传参的顺序可以不固定  
# fun01(c=30,a=10,b=20)  
# fun01(a=10,b=20,d=40)#错误  
  
#字典实参 使用\*\*将字典拆开，字典中的键值对以关键字的形式进行对应，传递值  
# dict01 = {'a':10,'b':20,'c':30}  
#a = 10 , b = 20 ,c = 30  
# fun01(\*\*dict01)  
#字典中的键的名字要与形参名对应  
# dict01 = {'a':10,'e':20,'d':30}  
# fun01(\*\*dict01)#报错  
  
# 混合使用  
# 语法规定 先写位置参数 再写关键字参数  
# fun01(10,20,c=30)  
# fun01(c=30,b=20,10)#报错  
# fun01(10,c=30,b=20)*

### 形参定义方式parameter

#### 缺省参数

1. 语法：

def 函数名(形参名1=默认实参1, 形参名2=默认实参2, ...):

函数体

1. 说明：

缺省参数必须自右至左依次存在，如果一个参数有缺省参数，则其右侧的所有参数都必须有缺省参数。

缺省参数可以有0个或多个，甚至全部都有缺省参数。

#### 位置形参

语法：

def 函数名(形参名1, 形参名2, ...):

函数体

##### 星号元组形参

1. 语法：

def 函数名(\*元组形参名):

  函数体

1. 作用：

收集多余的位置传参。

1. 说明：

一般命名为'args'

形参列表中最多只能有一个

#### 命名关键字形参

1. 语法：

def 函数名(\*, 命名关键字形参1, 命名关键字形参2, ...):

函数体

def 函数名(\*args, 命名关键字形参1, 命名关键字形参2, ...):  
  函数体

1. 作用：

强制实参使用关键字传参

##### 双星号字典形参

1. 语法：

def 函数名(\*\*字典形参名):

函数体

1. 作用：

收集多余的关键字传参

1. 说明:

一般命名为'kwargs'

形参列表中最多只能有一个

#### 参数自左至右的顺序

位置形参--> 星号元组形参-->命名关键字形参--> 双星号字典形参

*'''  
函数参数传递  
形参  
'''  
#打印矩形 要求传递行数和列数 还有填充字符  
#如果用户不传递填充字符 默认使用\*  
#默认形参 实参可以不传递数据 使用形参的默认值  
#当实参传递数据时 会覆盖默认值  
# def fun01(row,col,char='\*'):  
# for r in range(row):  
# for c in range(col):  
# print(char,end=' ')  
# print()  
#  
# fun01(3,3)  
# fun01(3,3,'-')  
  
# def fun01(a = 0,b = 1,c = 2):  
# print(a)  
# print(b)  
# print(c)  
#  
# #传参方式更灵活  
# fun01()  
# fun01(1,'b')  
# fun01(b='bbbb')  
  
#星号元组形参  
#接受不定数量的实参 让实参的数量可以无限多  
# def fun02(p1,p2,\*args):  
# print(p1)  
# print(p2)  
# print(\*args)  
#  
# fun02(1,2,3,4,5,6,7,8,9,10)  
# fun02(1,2)  
  
#在星号元组形参以后的形参叫做命名关键字形参  
#传递实参的过程中 必须指定形参的名字  
# def fun03(\*args,p1 = '',p2):  
# print(\*args)  
# print(p1)  
# print(p2)  
# fun03(1,2,3,4,5,6,p2=20)  
# 案例  
#print()  
#sep是在两个值中间输出的字符串  
#end是在最后一个值后面输出的字符串  
# def myprint(\*args,sep=' ',end='\n',file=None):  
# print(\*args,sep=sep,end=end)  
# myprint(1,2,3,4)  
# myprint(10,20,30,sep='+')  
#星号以后的位置形参是命名关键字形参  
#传递实参只接受关键字实参  
# def fun04(\*,p1=0,p2):  
# print(p1)  
# print(p2)  
#  
# fun04(p1=100,p2=200)  
# fun04(p2=200)  
  
#双星号字典形参 让关键字实参的数量无限  
#将关键字实参的变量名作为字典的键 值做为字典中对应键的值保存  
# def fun05(\*\*kwargs):  
# print(kwargs)  
#  
# fun05(a=10)  
# fun05(a=10,b=20,c=30)  
  
# 混合使用  
#语法 位置形参要在关键字形参前面定义  
#只要出现关键字形参 后面的参数也要带有关键字  
# def fun06(a=10,b,c):#报错  
# print(a)  
# print(b)  
# print(c)  
#  
# fun06(20,30)  
  
#定义一个函数 函数中包含位置形参、星号元组形参、默认形参、命名关键字形参和双星号字典形参  
# def fun07(a,\*args,b='bbb',c,\*\*kwargs):  
# print(a)  
# print(args)  
# print(b)  
# print(c)  
# print(kwargs)  
#  
# fun07(1,c='ccc')  
# fun07(5,6,7,c='abc',d='789')  
  
#定义一个函数 接受任意的参数 并输出结果  
# def fun08(\*args,\*\*kwargs):  
# for item in args:  
# print(item)  
# for k,v in kwargs.items():  
# print(k,v)  
#  
# fun08()  
# fun08(5,6,7,a=123,b='789')*

# 作用域LEGB

1. 作用域：变量起作用的范围。
2. Local局部作用域：函数内部。
3. Enclosing 外部嵌套作用域 ：函数嵌套。
4. Global全局作用域：模块(.py文件)内部。
5. Builtin内置模块作用域：builtins.py文件。

## 变量名的查找规则

1. 由内到外：L -> E -> G -> B
2. 在访问变量时，先查找本地变量，然后是包裹此函数外部的函数内部的变量，之后是全局变量，最后是内置变量。

## 局部变量

1. 定义在函数内部的变量(形参也是局部变量)
2. 只能在函数内部使用
3. 调用函数时才被创建，函数结束后自动销毁

## 全局变量

1. 定义在函数外部,模块内部的变量。
2. 在整个模块(py文件)范围内访问（但函数内不能将其直接赋值）。

## global 语句

1. 作用：

在函数内部修改全局变量。

在函数内部定义全局变量(全局声明)。

1. 语法：

global 变量1, 变量2, …

1. 说明

在函数内直接为全局变量赋值，视为创建新的局部变量。

不能先声明局部的变量，再用global声明为全局变量。

*#Global 全局作用域*a = 100 *#全局变量 在文件内都可以访问到***def** change():  
 *#Local 局部作用域  
 #对于inner\_fun来说 外部嵌套作用域* a = 10 *#局部变量 只能在函数内部使用  
 #当函数调用时创建局部变量 函数结束后销毁* **def** inner\_fun():  
 *# Local 局部作用域  
 # a = 1* print(a)  
 inner\_fun()  
 print(a)  
  
change()

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

*# g01 = "悟空"  
# g02 = "八戒"  
# def fun01():  
# num1 = 100  
# print(num1)  
#  
# g02 = "老朱"  
# print(g02)  
#  
# global g03  
# g03 = '老沙'  
#  
# #先定义了局部变量  
# #再声明全局变量 有歧义  
# #3.6版本后报错  
# # global g02  
# # g02 = '猪八戒'  
# # print(g02)  
#  
# fun01()  
# print(g03)  
# print(g02)  
  
# def fun01():  
# a = 10  
# def inner\_fun():  
# # a = 2  
# nonlocal a  
# a+=1#a = a + 1  
# print(a)  
# inner\_fun()  
# print(a)  
#  
# fun01()***def** user\_info():  
 uesr\_name = **'shibw'** user\_age = 20  
 user\_email = **'shibw@tedu.cn'** gender = **'男'** address = **'北京市东城区'** *# return (uesr\_name,user\_age,uest\_email,gender,address)  
 # return user\_age  
 # return {'uesr\_name':uesr\_name,'user\_age':user\_age,'user\_email':uesr\_email,'gender':gender,'address':address}  
 #locals() 收集当前的局部变量 保存到字典* **return** locals()  
  
*#获取到返回的元组*print(user\_info())  
inofs = user\_info()  
*#输出详细信息*print(**'姓名：%s,年龄：%s,邮箱：%s,性别：%s,地址：%s'** % (inofs[**'uesr\_name'**],inofs[**'user\_age'**],inofs[**'user\_email'**],inofs[**'gender'**],inofs[**'address'**]) )

## nonlocal 语句

1. 作用：

在内层函数修改外层嵌套函数内的变量

1. 语法

nonlocal 变量名1,变量名2, ...

1. 说明

在被嵌套的内函数中进行使用

# 面向对象 Object Oriented

## 概述

### 面向过程

1. 分析出解决问题的步骤，然后逐步实现。

例如：婚礼筹办

-- 发请柬（选照片、措词、制作）

-- 宴席（场地、找厨师、准备桌椅餐具、计划菜品、购买食材）

-- 婚礼仪式（定婚礼仪式流程、请主持人）

1. 公式：程序 = 算法 + 数据结构
2. 优点：所有环节、细节自己掌控。
3. 缺点：考虑所有细节，工作量大。

### 面向对象

1. 找出解决问题的人，然后分配职责。

例如：婚礼筹办

-- 发请柬：找摄影公司（拍照片、制作请柬）

-- 宴席：找酒店（告诉对方标准、数量、挑选菜品）

-- 婚礼仪式：找婚庆公司（对方提供司仪、制定流程、提供设备、帮助执行）

1. 公式：程序 = 对象 + 交互
2. 优点
3. 思想层面：

-- 可模拟现实情景，更接近于人类思维。

-- 有利于梳理归纳、分析解决问题。

1. 技术层面：

-- 高复用：对重复的代码进行封装，提高开发效率。

-- 高扩展：增加新的功能，不修改以前的代码。

-- 高维护：代码可读性好，逻辑清晰，结构规整。

1. 缺点：学习曲线陡峭。

## 类和对象

1. 类：一个抽象的概念，即生活中的”类别”。
2. 对象：类的具体实例，即归属于某个类别的”个体”。
3. 类是创建对象的”模板”。

-- 数据成员：名词类型的状态。

-- 方法成员：动词类型的行为。

1. 类与类行为不同，对象与对象数据不同。

### 语法

#### 定义类

1. 代码

class 类名:

“””文档说明”””

def \_init\_(self,参数列表):

self.实例变量 = 参数

方法成员

*'''*

*封装*

*'''*

*#数据角度:Wife(名字,身高,体重) Student(姓名,年龄,成绩)*

*#{姓名：[年龄,成绩]}*

**class** Wife:

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

self.name = name

*#私有成员：以双下划线开头*

*# self.\_\_age = age*

self.set\_age(age)

**def** get\_age(self):

**return** self.\_\_age

**def** set\_age(self,value):

**if** 25<=value<=30:

self.\_\_age = value

**else**:

*#抛出异常*

**raise** ValueError(**"我不要！"**)

w01 = Wife(**'甄宓'**,250)

*#不能访问私有变量*

*# print(w01.\_\_age)*

*# w01.set\_age(25)*

print(w01.get\_age())

1. 说明

-- 类名所有单词首字母大写.

-- \_init\_ 也叫构造函数，创建对象时被调用，也可以省略。

-- **self 变量绑定的是被创建的对象**，名称可以随意。

#### 创建对象(实例化)

变量 = 构造函数 (参数列表)

### 实例成员

#### 实例变量

1. 语法
2. 定义：对象.变量名
3. 调用：对象.变量名
4. 说明
5. 首次通过对象赋值为创建，再次赋值为修改.

w01 = Wife()

w01.name = “丽丽”

w01.name = “莉莉”

1. 通常在构造函数(\_init\_)中创建。

w01 = Wife(“丽丽”,24)

print(w01.name)

1. 每个对象存储一份，通过对象地址访问。
2. 作用：描述某个对象的数据。
3. \_\_dict\_\_：对象的属性，用于存储自身实例变量的字典。

#### 实例方法

1. 语法

(1) 定义： def 方法名称(self, 参数列表):

方法体

(2) 调用： 对象地址.实例方法名(参数列表)

不建议通过类名访问实例方法

1. 说明

(1) 至少有一个形参，第一个参数绑定调用这个方法的对象,一般命名为"self"。

(2) 无论创建多少对象，方法只有一份，并且被所有对象共享。

1. 作用：表示对象行为。

### 类成员

#### 类变量

1. 语法
2. 定义：在类中，方法外定义变量。

class 类名:

变量名 = 表达式

1. 调用：类名.变量名

不建议通过对象访问类变量

1. 说明

(1) 存储在类中。

(2) 只有一份，被所有对象共享。

1. 作用：描述所有对象的共有数据。

#### 类方法

1. 语法
2. 定义：

@classmethod

def 方法名称(cls,参数列表):

方法体

1. 调用：类名.方法名(参数列表)

不建议通过对象访问类方法

1. 说明

(1) 至少有一个形参，第一个形参用于绑定类，一般命名为'cls'

(2) 使用@classmethod修饰的目的是调用类方法时可以隐式传递类。

(3) 类方法中不能访问实例成员，实例方法中可以访问类成员。

1. 作用：操作类变量。

**class** ICBC:

*#类变量 总行的钱*

total\_money = 1000000

*#类方法*

@classmethod

*#cls保存当前类的地址*

**def** print\_total\_money(cls):

print(id(cls),id(ICBC))

print(cls.total\_money)

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

self.name = name

self.money = money

*#从总行扣除钱数*

ICBC.total\_money -= money

*#查询/操作类中的数据 不要用实例方法*

*# def print\_total\_money(self):*

*# print(ICBC.total\_money)*

i01 = ICBC(**'北京天坛支行'**,100000)

i02 = ICBC(**'北京万寿路支行'**,100000)

ICBC.print\_total\_money()

*# 非主流：通过对象访问类成员*

*# i02.print\_total\_money()*

### 静态方法

1. 语法
2. 定义：

@staticmethod

def 方法名称(参数列表):

方法体

1. 调用：类名.方法名(参数列表)

不建议通过对象访问静态方法

1. 说明

(1) 使用@ staticmethod修饰的目的是该方法不需要隐式传参数。

(2) 静态方法不能访问实例成员和类成员

1. 作用：定义常用的工具函数。

## 三大特征

*# 面向对象三大特征*

*# 封装*

*# 语法：*

*# 数据：一个类包装多个变量*

*# 行为：提供必要的功能 隐藏实现的细节*

*# 设计：分*

*# 继承*

*# 语法：子类拥有父类的所有成员*

*# 设计：隔离变化*

*# 多态*

*# 语法：重写（子覆盖父类）*

*# 作法：父类 创建子类对象 重写* *-->调用父执行子*

*# 设计：做*

*#类与类的关系：*

*#组合(实例变量)/泛化(继承)/依赖(参数)*

*#面向对象的设计原则：*

*#开闭原则：允许增加，不允许修改*

*#单一职责：一个变化*

*#依赖倒置：调用父，不调用子*

*#组合复用：通过变量(参数/实例变量)调用 而不是使用继承*

*#里氏替换：扩展重写*

*#迪米特法则：低耦合*

### 封装

#### 数据角度讲

1. 定义：

将一些基本数据类型复合成一个自定义类型。

1. 优势：

将数据与对数据的操作相关联。

代码可读性更高（类是对象的模板）。

#### 行为角度讲

1. 定义：

类外提供必要的功能，隐藏实现的细节。

1. 优势：

简化编程，使用者不必了解具体的实现细节，只需要调用对外提供的功能。

1. 私有成员：
2. 作用：无需向类外提供的成员，可以通过私有化进行屏蔽。
3. 做法：命名使用双下划线开头。
4. 本质：障眼法，实际也可以访问。

私有成员的名称被修改为：\_类名\_\_成员名，可以通过\_dict\_属性或dir函数查看。

1. 属性@property：

公开的实例变量，缺少逻辑验证。私有的实例变量与两个公开的方法相结合，又使调用者的操作略显复杂。而属性可以将两个方法的使用方式像操作变量一样方便。

1. 定义：

@property

def 属性名(self):

return self.\_\_属性名

@属性名.setter

def 属性名(self, value):

self.\_\_属性名= value

1. 调用：

对象.属性名 = 数据

变量 = 对象.属性名

1. 说明：

通常两个公开的属性，保护一个私有的变量。

@property 负责读取，@属性名.setter 负责写入

只写：属性名= property(None, 写入方法名)

*'''*

*封装*

*'''*

*#数据角度:Wife(名字,身高,体重) Student(姓名,年龄,成绩)*

*#{姓名：[年龄,成绩]}*

*#创建敌人类 （名字 血量0--100 攻击力1-50）*

**class** Wife:

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

self.name = name

*#私有成员：以双下划线开头*

*# self.\_\_age = age*

*# self.set\_age(age)*

self.age = age

@property *#拦截读取* *age = property(get\_age,None)*

**def** age(self):

**return** self.\_\_age

*#设置写入方法age.setter(写入方法)*

@age.setter *#拦截写入*

**def** age(self,value):

**if** 25<=value<=30:

self.\_\_age = value

**else**:

*#抛出异常*

**raise** ValueError(**"我不要！"**)

*#property 拦截对age的读写操作*

*#age = property(get\_age,set\_age)*

w01 = Wife(**'甄宓'**,25)

*#不能访问私有变量*

*# print(w01.\_\_age)*

*# w01.set\_age(25)*

*# print(w01.get\_age())*

print(w01.age)

print(w01.\_\_dict\_\_)

print(w01.\_Wife\_\_age)

#### 设计角度讲

1. 定义：

(1) 分而治之

将一个大的需求分解为许多类，每个类处理一个独立的功能。

(2) 变则疏之

变化的地方独立封装，避免影响其他类。

(3) 高 内 聚

类中各个方法都在完成一项任务(单一职责的类)。

(4) 低 耦 合

类与类的关联性与依赖度要低(每个类独立)，让一个类的改变，尽少影响其他类。

1. 优势：

便于分工，便于复用，可扩展性强。

#### 案例:信息管理系统



##### 需求

实现对学生信息的增加、删除、修改和查询。

##### 分析

界面可能使用控制台，也可能使用Web等等。

1. 识别对象：界面视图类 逻辑控制类 数据模型类
2. 分配职责：

界面视图类：负责处理界面逻辑，比如显示菜单，获取输入，显示结果等。

逻辑控制类：负责存储学生信息，处理业务逻辑。比如添加、删除等

数据模型类：定义需要处理的数据类型。比如学生信息。

1. 建立交互：

界面视图对象 <----> 数据模型对象 <----> 逻辑控制对象

##### 设计

数据模型类：StudentModel

数据：编号 id,姓名 name,年龄 age,成绩 score

逻辑控制类：StudentManagerController

数据：学生列表 \_\_stu\_list

行为：获取列表 stu\_list,添加学生 add\_student，删除学生remove\_student，修改学生update\_student，根据成绩排序order\_by\_score。

界面视图类：StudentManagerView

数据：逻辑控制对象\_\_manager

行为：显示菜单\_\_display\_menu，选择菜单项\_\_select\_menu\_item，入口逻辑main，

输入学生\_\_input\_students，输出学生\_\_output\_students，删除学生\_\_delete\_student，修改学生信息\_\_modify\_student

### 继承

#### 语法角度讲

##### 继承方法

1. 代码:

class 父类:

def 父类方法(self):

方法体

class 子类(父类)：

def 子类方法(self):

方法体

儿子 = 子类()

儿子.子类方法()

儿子.父类方法()

1. 说明：

子类直接拥有父类的方法.

*'''*

*继承 多态*

*'''*

*# 老张去东北*

*# 需求变化 ： 坐飞机*

*# 坐火车*

*# 骑车*

**class** Vehicle:

*#交通工具 代表所有的具体的交通工具*

*#*

**def** transport(self,position):

*#因为父类太抽象，写不出具体的方法体*

**pass**

**class** Person:

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

self.name = name

**def** go\_to(self,position,vehicle):

print(**'去：'** + position)

vehicle.transport(position)

*# ----------------------------------------*

**class** Car(Vehicle):

**def** transport(self,position):

print(**'开车到'**,position)

**class** Airplane(Vehicle):

**def** transport(self,position):

print(**'飞到'**,position)

c01 = Car()

a01 = Airplane()

lz = Person(**'老张'**)

lz.go\_to(**'东北'**, c01)

lz.go\_to(**'东北'**, a01)

*# 定义一个图形管理器类*

*# 1.保存所有的图形*

*# 2.提供计算 所有图形总面积的功能*

**class** GraphicManager:

**def** \_\_init\_\_(self):

self.\_\_list\_graphic = []

@property

**def** list\_graphic(self):

**return** self.\_\_list\_graphic

**def** add\_graphic(self,graphic):

**if not** isinstance(graphic,Graphic):

**raise** ValueError()

**else**:

self.\_\_list\_graphic.append(graphic)

**def** calc\_total\_area(self):

total\_area = 0

**for** item **in** self.\_\_list\_graphic:

total\_area += item.calc\_area()

**return** total\_area

**class** Graphic:

**def** calc\_area(self):

*#如果子类不重写父类的功能 异常*

**raise** NotImplementedError()

*# 圆形* *pi\*r\*\*2*

*# 矩形 长\*宽*

*# 要求增加新图形时 不影响管理器代码*

**class** Circle(Graphic):

**def** \_\_init\_\_(self, radius):

self.radius = radius

**def** calc\_area(self):

**return** 3.14 \* self.radius \*\* 2

**class** Rectangle(Graphic):

**def** \_\_init\_\_(self, length, width):

self.length = length

self.width = width

**def** calc\_area(self):

**return** self.length\*self.width

c01 = Circle(5)

r01 = Rectangle(10,20)

manager = GraphicManager()

manager.add\_graphic(c01)

manager.add\_graphic(r01)

*# print(manager.list\_graphic)*

print(manager.calc\_total\_area())

##### 内置函数

isinstance(对象, 类型)

返回指定对象是否是某个类的对象。

issubclass(类型，类型)

返回指定类型是否属于某个类型。

##### 继承数据

1. 代码

class 子类(父类):

def \_\_init\_\_(self,参数列表):

super().\_\_init\_\_(参数列表)

self.自身实例变量 = 参数

1. 说明

子类如果没有构造函数，将自动执行父类的，但如果有构造函数将覆盖父类的。此时必须通过super()函数调用父类的构造函数，以确保父类实例变量被正常创建。

##### 定义

重用现有类的功能，并在此基础上进行扩展。

说明：子类直接具有父类的成员（共性），还可以扩展新功能。

##### 优点

一种代码复用的方式。

##### 缺点

耦合度高：父类的变化，直接影响子类。

#### 设计角度讲

##### 定义

将相关类的共性进行抽象，统一概念，隔离变化。

##### 适用性

多个类在概念上是一致的，且需要进行统一的处理。

##### 相关概念

父类（基类、超类）、子类（派生类）。

父类相对于子类更抽象，范围更宽泛；子类相对于父类更具体，范围更狭小。

单继承：父类只有一个（例如 Java，C#）。

多继承：父类有多个（例如C++，Python）。

Object类：任何类都直接或间接继承自 object 类。

#### 多继承

一个子类继承两个或两个以上的基类，父类中的属性和方法同时被子类继承下来。

同名方法的解析顺序（MRO， Method Resolution Order）:

类自身 --> 父类继承列表（由左至右）--> 再上层父类

A

/ \

/ \

B C

\ /

\ /

D

### 多态

#### 设计角度讲

##### 定义

父类的同一种动作或者行为，在不同的子类上有不同的实现。

##### 作用

1. 在继承的基础上，体现类型的个性化（一个行为有不同的实现）。
2. 增强程序扩展性，体现开闭原则。

#### 语法角度讲

##### 重写

子类实现了父类中相同的方法（方法名、参数）。

在调用该方法时，实际执行的是子类的方法。

##### 快捷键

Ctrl + O

##### 内置可重写函数

Python中，以双下划线开头、双下划线结尾的是系统定义的成员。我们可以在自定义类中进行重写，从而改变其行为。

###### 转换字符串

\_\_str\_\_函数：将对象转换为字符串(对人友好的)

\_\_repr\_\_函数：将对象转换为字符串(解释器可识别的)

###### 运算符重载

定义：让自定义的类生成的对象(实例)能够使用运算符进行操作。

算数运算符



反向算数运算符重载



复合运算符重载



比较运算重载



## 设计原则

### 开-闭原则（目标、总的指导思想）

**O**pen **C**losed **P**rinciple

对扩展开放，对修改关闭。

增加新功能，不改变原有代码。

### 类的单一职责（一个类的定义）

**S**ingle **R**esponsibility **P**rinciple

一个类有且只有一个改变它的原因。

### 依赖倒置（依赖抽象）

**D**ependency **I**nversion **P**rinciple

客户端代码(调用的类)尽量依赖(使用)抽象。

抽象不应该依赖细节，细节应该依赖抽象。

### 组合复用原则（复用的最佳实践）

Composite Reuse Principle

如果仅仅为了代码复用优先选择组合复用，而非继承复用。

组合的耦合性相对继承低。

### 里氏替换（继承后的重写，指导继承的设计）

**L**iskov **S**ubstitution **P**rinciple

父类出现的地方可以被子类替换，在替换后依然保持原功能。

子类要拥有父类的所有功能。

子类在重写父类方法时，尽量选择扩展重写，防止改变了功能。

### 迪米特法则（类与类交互的原则）

Law of Demeter

不要和陌生人说话。

类与类交互时，在满足功能要求的基础上，传递的数据量越少越好。因为这样可能降低耦合度。

## **时间戳**

*'''*

*时间戳*

*'''*

**import** demo03

*#获取当前时间戳 从1970年1月1日到现在的总秒数*

*# print(time.time())*

*#获取当前的时间元组*

*#(年 月 日 时 分 秒 一周的第几天(0 1 2...) 一年的第几天 夏令时)*

*# print(time.localtime())*

*#获取指定时间戳的时间元组*

*# print(time.localtime(12345678))*

time\_tuple = demo03.localtime()

*#将时间元组变成时间戳*

*# print(time.mktime(time\_tuple))*

*#将时间元组转换成时间字符串*

*# print(time.strftime('%y/%m/%d %H:%M:%S',time\_tuple))*

*# print(time.strftime('%Y-%m-%d %H:%M:%S',time\_tuple))*

*#将日期字符串转换为时间元组*

print(demo03.strptime(**"19/08/21 11:31:30"**, **'%y/%m/%d %H:%M:%S'**))

print(demo03.strptime(**"2019/08/21 11:31:30"**, **'%Y/%m/%d %H:%M:%S'**))

# 模块 Module

## 定义

包含一系列数据、函数、类的文件，通常以.py结尾。

## 作用

让一些相关的数据，函数，类有逻辑的组织在一起，使逻辑结构更加清晰。

有利于多人合作开发。

## 导入

### import

1. 语法：

import 模块名

import 模块名 as 别名

1. 作用：将某模块整体导入到当前模块中
2. 使用：模块名.成员

### from import

1. 语法：

from 模块名 import 成员名[ as 别名1]

作用：将模块内的一个或多个成员导入到当前模块的作用域中。

### from import \*

1. 语法：from 模块名 import \*
2. 作用：将某模块的所有成员导入到当前模块。
3. 模块中以下划线(\_)开头的属性，不会被导入，通常称这些成员为隐藏成员。

**模块变量**

\_\_all\_\_变量：定义可导出成员，仅对from xx import \*语句有效。

\_\_doc\_\_变量：文档字符串。

\_\_file\_\_变量：模块对应的文件路径名。

\_\_name\_\_变量：模块自身名字，可以判断是否为主模块。

当此模块作为主模块(第一个运行的模块)运行时，\_\_name\_\_绑定'\_\_main\_\_'，不是主模块，而是被其它模块导入时,存储模块名。

## 加载过程

在模块导入时，模块的所有语句会执行。

如果一个模块已经导入，则再次导入时不会重新执行模块内的语句。

## 分类

1. 内置模块(builtins)，在解析器的内部可以直接使用。
2. 标准库模块，安装Python时已安装且可直接使用。
3. 第三方模块（通常为开源），需要自己安装。
4. 用户自己编写的模块（可以作为其他人的第三方模块）

*#定义函数 根据年月日计算星期几*

*#时间元组*

**import** time

**def** get\_week(year, month, day):

time\_tuple=time.strptime(**'%d/%d/%d'** % (year, month, day), **'%Y/%m/%d'**)

week\_tuple=(**'星期一'**, **'星期二'**, **'星期三'**, **'星期四'**,

**'星期五'**, **'星期六'**, **'星期日'**)

print(time\_tuple)

**return** week\_tuple[time\_tuple[6]]

**def** life\_days(bir):

time\_tuple=time.strptime(bir, **'%Y/%m/%d'**)

*# 生活总秒数=当前时间戳-出生时间戳*

life\_sceond=time.time() - time.mktime(time\_tuple)

**return** life\_sceond / 60 / 60 // 24 // 365

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

print(get\_week(2019, 8, 21))

print(**'这是第%d个生日'** % life\_days(**'1990/6/5'**))

## 搜索顺序

搜索内建模块(builtins)

sys.path 提供的路径，通常第一个是程序运行时的路径。

# 包package

## 定义

将模块以文件夹的形式进行分组管理。

## 作用

让一些相关的模块组织在一起，使逻辑结构更加清晰。

## 导入

import 包名 [as 包别名] 需要设置\_\_all\_\_

import 包名.模块名 [as 模块新名]

import 包名.子包名.模块名 [as 模块新名]

from 包名 import 模块名 [as 模块新名]

from 包名.子包名 import 模块名 [as 模块新名]

from 包名.子包名.模块名 import 成员名 [as 属性新名]

# 导入包内的所有子包和模块

from 包名 import \*

from 包名.模块名 import \*

## 搜索顺序

sys.path 提供的路径

## \_\_init\_\_.py 文件

是包内必须存在的文件

会在包加载时被自动调用

### \_\_all\_\_

记录from 包 import \* 语句需要导入的模块

案例：

my\_ project /

main.py

common/

\_\_init\_\_.py

double\_list\_helper.py

list\_helper.py

skill\_system/

\_\_init\_\_.py

skill\_deployer.py

skill\_manager.py

练习:

1. 在main.py中调用skill\_deployer.py。
2. 在skill\_deployer.py中调用skill\_manager.py。
3. 在skill\_manager.py中调用double\_list\_helper.py。
4. 在list\_helper.py中调用main.py。

要求：在所有的调用过程中，要包含函数、类、实例方法、静态方法。

14：45

# 异常处理Error

## 异常

1. 定义：运行时检测到的错误。
2. 现象：当异常发生时，程序不会再向下执行，而转到函数的调用语句。
3. 常见异常类型：

-- 名称异常(NameError)：变量未定义。

-- 类型异常(TypeError)：不同类型数据进行运算。

-- 索引异常(IndexError)：超出索引范围。

-- 属性异常(AttributeError)：对象没有对应名称的属性。

-- 键异常(KeyError)：没有对应名称的键。

-- 为实现异常(NotImplementedError)：尚未实现的方法。

-- 异常基类Exception。

## 处理

1. 语法：

try:

可能触发异常的语句

except 错误类型1 [as 变量1]：

处理语句1

except 错误类型2 [as 变量2]：

处理语句2

except Exception [as 变量3]：

不是以上错误类型的处理语句

else:

未发生异常的语句

finally:

无论是否发生异常的语句

1. 作用：将程序由异常状态转为正常流程。
2. 说明：

as 子句是用于绑定错误对象的变量，可以省略

except子句可以有一个或多个，用来捕获某种类型的错误。

else子句最多只能有一个。

finally子句最多只能有一个，如果没有except子句，必须存在。

如果异常没有被捕获到，会向上层(调用处)继续传递，直到程序终止运行。

**def** div\_apple(apple\_count):

person\_count=int(input(**"请输入人数:"**))

res=apple\_count/person\_count

print(**"每个人分到了%s个苹果"**% res)

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

**try**:

*# 可能出现异常的语句*

div\_apple(10)

**except** ValueError:

print(**'人数必须为整数'**)

**except** ZeroDivisionError:

print(**"人数不能为0"**)

**except** KeyboardInterrupt:

print(**'\n用户中断输入'**)

**except** Exception:

print(**'未知错误'**)

**else**:

*# 没有发生异常执行的语句*

print(**'没有异常发生'**)

**finally**:

*# 不管有没有发生异常一定要执行的语句*

print(**'程序结束'**)

## raise 语句

1. 作用：抛出一个错误，让程序进入异常状态。
2. 目的：在程序调用层数较深时，向主调函数传递错误信息要层层return 比较麻烦，所以人为抛出异常，可以直接传递错误信息。。

## 自定义异常

1. 定义：

class 类名Error(Exception):

def \_\_init\_\_(self,参数):

super().\_\_init\_\_(参数)

self.数据 = 参数

*# 自定义异常类*

**class** AgeError(Exception):

**def** \_\_init\_\_(self, message, code,id):

*# 错误信息*

self.message=message

*# 错误代码*

self.code=code

*# 错误ID*

self.id=id

1. 调用：

try:

….

raise 自定义异常类名(参数)

….

except 定义异常类 as 变量名:

变量名.数据

**def** get\_score():

**while True**:

**try**:

score=float(input(**"请输入成绩:"**))

**return** score

**except** :

print(**'输入有误'**)

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

print(get\_score())

1. 作用：封装错误信息

# 迭代

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。例如：循环获取容器中的元素。

## 可迭代对象iterable

1. 定义：具有\_\_iter\_\_函数的对象，可以返回迭代器对象。
2. 语法

-- 创建：

class 可迭代对象名称:

  def \_\_iter\_\_(self):

      return 迭代器

1. 原理：

迭代器 = 可迭代对象.\_\_iter\_\_()

while True:

try:

print(迭代器.\_\_next\_\_())

except StopIteration:

break

**touple01=(4, 5, 66, 7, 2)**

**for** i **in** range(len(touple01)):

print(touple01[i])

dict01={

**'张翠山'**: 101, **'殷素素'**: 102, **'张无忌'**: 103

}

iterator=dict01.\_\_iter\_\_()

**while True**:

**try**:

key=iterator.\_\_next\_\_()

print(key,dict01[key])

**except** StopIteration:

**break**

## 迭代器对象iterator

1. 定义：可以被next()函数调用并返回下一个值的对象。
2. 语法

class 迭代器类名:

def \_\_init\_\_(self, 聚合对象):

self.聚合对象= 聚合对象

def \_\_next\_\_(self):

if 没有元素:

raise StopIteration

return 聚合对象元素

1. 说明：

-- 聚合对象通常是容器对象。

4. 作用：使用者只需通过一种方式，便可简洁明了的获取聚合对象中各个元素，而又无需了解其内部结构。

*"""*

*迭代器* *--> yield*

*目标：让自定义类所创建的对象，可以参与for.*

*iter价值：可以被for*

*next价值：返回数据/抛出异常*

*class 自定义类的迭代器：*

*def \_\_next\_\_(self):*

*pass*

*class 自定义类:*

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

*pass*

*for item in 自定义类():*

*pass*

*"""*

*# class SkillIterator:*

*# def \_\_init\_\_(self,data):*

*# self.\_\_target = data*

*# self.\_\_index = -1*

*#*

*# def \_\_next\_\_(self):*

*# # 如果没有数据则抛出异常*

*# if self.\_\_index >= len(self.\_\_target)-1:*

*# raise StopIteration*

*# # 返回数据*

*# self.\_\_index += 1*

*# return self.\_\_target[self.\_\_index]*

**class** SkillManager:

*"""*

*技能管理器 可迭代对象*

*"""*

**def** \_\_init\_\_(self):

self.\_\_skills = []

**def** add\_skill(self,str\_skill):

self.\_\_skills.append(str\_skill)

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

*# return SkillIterator(self.\_\_skills)*

*# 执行过程：*

*# 1. 调用\_\_iter\_\_()不执行*

*# 2. 调用\_\_next\_\_()才执行当前代码*

*# 3. 执行到yield语句暂时离开*

*# 4. 再次调用\_\_next\_\_()继续执行*

*# ....*

*# yield作用：标记着下列代码会自动转换为迭代器代码.*

*# 转换大致过程：*

*# 1. 将yield关键字以前的代码，放到next方法中。*

*# 2. 将yield关键字后面的数据,作为next返回值.*

*# print("准备数据：")*

*# yield "降龙十八掌"*

*#*

*# print("准备数据：")*

*# yield "黑虎掏心"*

*#*

*# print("准备数据：")*

*# yield "六脉神剑"*

**for** item **in** self.\_\_skills:

**yield** item

manager = SkillManager()

manager.add\_skill(**"降龙十八掌"**)

manager.add\_skill(**"黑虎掏心"**)

manager.add\_skill(**"六脉神剑"**)

*# 错误：manager必须是可迭代对象\_\_iter\_\_(),*

*# for item in manager:*

*# print(item)*

iterator = manager.\_\_iter\_\_()

**while True**:

**try**:

item = iterator.\_\_next\_\_()

print(item)

**except** StopIteration:

**break**

# 生成器generator

1. 定义：能够动态(循环一次计算一次返回一次)提供数据的可迭代对象。
2. 作用：在循环过程中，按照某种算法推算数据，不必创建容器存储完整的结果，从而节省内存空间。数据量越大，优势越明显。
3. 以上作用也称之为延迟操作或惰性操作，通俗的讲就是在需要的时候才计算结果，而不是一次构建出所有结果。

## 生成器函数

1. 定义：含有yield语句的函数，返回值为生成器对象。
2. 语法

-- 创建：

def 函数名():

…

yield 数据

…

-- 调用：

for 变量名 in 函数名():

语句

*"""*

*yield --> 生成器函数*

*"""*

*# class MyRange:*

*# def \_\_init\_\_(self,stop\_value):*

*# self.\_\_stop\_value= stop\_value*

*#*

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

*# number = -1*

*# while number < self.\_\_stop\_value -1 :*

*# number +=1*

*# yield number*

*#*

*#*

*# for item in MyRange(7):*

*# print(item)*

**def** my\_range(stop):

number = -1

**while** number < stop -1 :

number +=1

**yield** number

*# 惰性操作/延迟操作*

*# 返回值是生成器对象(可迭代对象* *+ 迭代器对象)*

iterator = my\_range(999)

**for** item **in** iterator:

print(item)

1. 说明：

-- 调用生成器函数将返回一个生成器对象，不执行函数体。

-- yield翻译为”产生”或”生成”

1. 执行过程：
2. 调用生成器函数会自动创建迭代器对象。
3. 调用迭代器对象的\_\_next\_\_()方法时才执行生成器函数。
4. 每次执行到yield语句时返回数据，暂时离开。
5. 待下次调用\_\_next\_\_()方法时继续从离开处继续执行。
6. 原理：生成迭代器对象的大致规则如下

-- 将yield关键字以前的代码放在next方法中。

-- 将yield关键字后面的数据作为next方法的返回值。

## 内置生成器

### 枚举函数enumerate

1. 语法：

for 变量 in enumerate(可迭代对象):

语句

for 索引, 元素in enumerate(可迭代对象):

语句

1. 作用：遍历可迭代对象时，可以将索引与元素组合为一个元组。

### zip

1. 语法：

for item in zip(可迭代对象1, 可迭代对象2….):

语句

1. 作用：将多个可迭代对象中对应的元素组合成一个个元组，生成的元组个数由最小的可迭代对象决定。

## 生成器表达式

1. 定义：用推导式形式创建生成器对象。
2. 语法：变量 = ( 表达式 for 变量 in 可迭代对象 [if 真值表达式] )

*"""*

*生成器表达式*

*"""*

list01 = [34,4,**"a"**,**"b"**,1.5,1.8,**True**,**False**]

*# 生成器函数:为其他人提供功能*

**def** find01():

**for** item **in** list01:

**if** type(item) == str:

**yield** item

re = find01()

**for** item **in** re:

print(item)

*# 生成器表达式：为自己提供功能*

re = (item **for** item **in** list01 **if** type(item) == str)

**for** item **in** re:

print(re)

# 函数式编程

1. 定义：用一系列函数解决问题。

-- 函数可以赋值给变量，赋值后变量绑定函数。

-- 允许将函数作为参数传入另一个函数。

-- 允许函数返回一个函数。

2. 高阶函数：将函数作为参数或返回值的函数。

## 函数作为参数

将核心逻辑传入方法体，使该方法的适用性更广，体现了面向对象的开闭原则。

### lambda 表达式

1. 定义：是一种匿名方法。
2. 作用：作为参数传递时语法简洁，优雅，代码可读性强。

随时创建和销毁，减少程序耦合度。

1. 语法

-- 定义：

变量 = lambda 形参: 方法体

-- 调用：

变量(实参)

1. 说明：

-- 形参没有可以不填

-- 方法体只能有一条语句，且不支持赋值语句。

*"""*

*lambda : 匿名函数*

*作用:充当实参*

*"""*

**def** fun01():

print(**"fun01"**)

fun01()

*# 无参数 无返回值*

a = **lambda** :print(**"fun01"**)

a()

**def** fun02(func):

print(**"fun02"**)

func()

*# 将函数作为参数,建议使用lambda.*

fun02(**lambda** :print(**"fun01"**))

**def** fun03(a,b,c):

print(**"fun03"**)

*# 有参数lambda*

b = **lambda** a,b,c:print(**"fun03"**)

**def** fun04():

print(**"fun04"**)

print(**"fun04又执行喽"**)

*# lambda 函数体只能有一句话*

*# fun02(lambda :print("fun04++");print("fun04又执行喽"))*

*# 不支持赋值语句*

*# fun02(lambda a:a.name = "zs")*

### yield

*"""*

*函数式编程 应用* *-- 将函数作为参数*

*"""*

list01 = [4,5,5,65,5,6]

**def** fun01():

**for** item **in** list01:

**if** item % 2:

**yield** item

**def** fun02():

**for** item **in** list01:

**if** item > 10:

**yield** item

**def** fun03():

**for** item **in** list01:

**if** 3 <item < 10:

**yield** item

*# 面向函数：*

*# "封装"：分*

**def** condition01(item):

**return** item % 2 == 1

**def** condition02(item):

**return** item > 10

**def** condition03(item):

**return** 3 <item < 10

*# "继承"：隔(通过参数，抽象具体函数。)*

*# “万能查找*”

**def** fun(func\_condition):

**for** item **in** list01:

*# if 3 <item < 10:*

*# if condition03(item):*

*# "多态"：做*

**if** func\_condition(item):

**yield** item

fun(condition01)

### 内置高阶函数

1. map（函数，可迭代对象）：使用可迭代对象中的每个元素调用函数，将返回值作为新可迭代对象元素；返回值为新可迭代对象。
2. filter(函数，可迭代对象)：根据条件筛选可迭代对象中的元素，返回值为新可迭代对象。
3. sorted(可迭代对象，key = 函数,reverse = bool值)：排序，返回值为排序结果。
4. max(可迭代对象，key = 函数)：根据函数获取可迭代对象的最大值。
5. min(可迭代对象，key = 函数)：根据函数获取可迭代对象的最小值。

*"""*

*内置高阶函数*

*"""*

**from** common.list\_helper **import** ListHelper

**class** Enemy:

**def** \_\_init\_\_(self, name, hp,atk=**None**, defense=**None**):

self.name = name

self.hp = hp

self.atk = atk

self.defense = defense

enemy\_list = [

Enemy(**'玄冥二老'**,86,80,120),

Enemy(**'成昆'**,0,0,150),

Enemy(**'谢逊'**,120,50,150),

Enemy(**'灭霸'**,0,0,999)

]

*# 1. 在敌人列表中查找所有死人*

**for** item **in** filter(**lambda** item:item.hp == 0,enemy\_list):

print(item.name)

*# for item in ListHelper.find\_all(enemy\_list,lambda item:item.hp == 0)*

*# 2.在敌人列表中查找所有敌人的名称*

**for** item **in** map(**lambda** item:item.name,enemy\_list):

print(item)

*# for item in ListHelper.select(enemy\_list,lambda item:item.name)*

*# 3. 获取攻击力最大的敌人*

re = max(enemy\_list,key = **lambda** item:item.atk )

print(re.name)

*# ListHelper.get\_max(enemy\_list,lambda item:item.atk )*

*# 4. 对敌人列表根据攻击力升序排列*

*# sorted 返回排好序的数据*

*# for item in sorted(enemy\_list,key = lambda item:item.atk):*

*# print(item.atk)*

*# 对敌人列表根据攻击力降序排列*

*# sorted 返回排好序的数据*

**for** item **in** sorted(enemy\_list,key = **lambda** item:item.atk,reverse=**True**):

print(item.atk)

ListHelper.order\_by(enemy\_list,**lambda** item:item.atk)

## 函数作为返回值

逻辑连续，当内部函数被调用时，不脱离当前的逻辑。

### 闭包

1. 三要素：

-- 必须有一个内嵌函数。

-- 内嵌函数必须引用外部函数中变量。

-- 外部函数返回值必须是内嵌函数。

1. 语法

-- 定义：

def 外部函数名(参数):

外部变量

def 内部函数名(参数):

使用外部变量

return 内部函数名

-- 调用：

变量 = 外部函数名(参数)

变量(参数)

1. 定义：在一个函数内部的函数,同时内部函数又引用了外部函数的变量。
2. 本质：闭包是将内部函数和外部函数的执行环境绑定在一起的对象。
3. 优点：内部函数可以使用外部变量。
4. 缺点：外部变量一直存在于内存中，不会在调用结束后释放，占用内存。
5. 作用：实现python装饰器。

*"""*

*闭包:外部函数执行完毕后,不立即释放内存.*

*而是等着内部函数使用外部嵌套变量.*

*练习:exercise03*

*"""*

**def** fun01():

a = 10

**def** fun02():

print(a)

**return** fun02*# 返回内部函数(没有执行)*

*# re 存储的是fun02*

re = fun01()

re()*# 调用fun02 10*

### 函数装饰器decorators

1. 定义：在不改变原函数的调用以及内部代码情况下，为其添加新功能的函数。
2. 语法

def 函数装饰器名称(func):

def 内嵌函数(\*args, \*\*kwargs):

需要添加的新功能

return func(\*args, \*\*kwargs)

return wrapper

@ 函数装饰器名称

def 原函数名称(参数):

函数体

原函数(参数)

1. 本质：使用“@函数装饰器名称”修饰原函数，等同于创建与原函数名称相同的变量，关联内嵌函数；故调用原函数时执行内嵌函数。

原函数名称 = 函数装饰器名称（原函数名称）

1. 装饰器链：

一个函数可以被多个装饰器修饰，执行顺序为从近到远。

*"""*

*装饰器*

*练习:exercise04.py*

*"""*

*# def print\_func\_name(func):*

*# def wrapper():*

*# print(func.\_\_name\_\_)# 打印函数名称*

*# func()# 调用函数*

*# return wrapper*

*# @print\_func\_name*

*# def say\_hello():*

*# print("hello")*

*# 拦截:新功能* *+ 旧功能*

*# say\_hello = print\_func\_name(say\_hello)*

*# def say\_goodbye():*

*# print("goodbye")*

*#*

*# say\_hello()*

*# say\_goodbye()*

*# 需求:在不改变原函数以及调用情况下,增加新功能(打印函数名称).*

**def** print\_func\_name(func):

*# \*args 原函数参数可以无限制*

**def** wrapper(\*args,\*\*kwargs):

print(func.\_\_name\_\_)*# 打印函数名称*

*# return 原函数返回值*

**return** func(\*args,\*\*kwargs)*# 调用函数*

**return** wrapper

@print\_func\_name

**def** say\_hello():

print(**"hello"**)

**return** 1

@print\_func\_name

**def** say\_goodbye(name):

print(name,**"---goodbye"**)

**return** 2

print(say\_hello())*#1*

print(say\_goodbye(**"qtx"**))*#2*