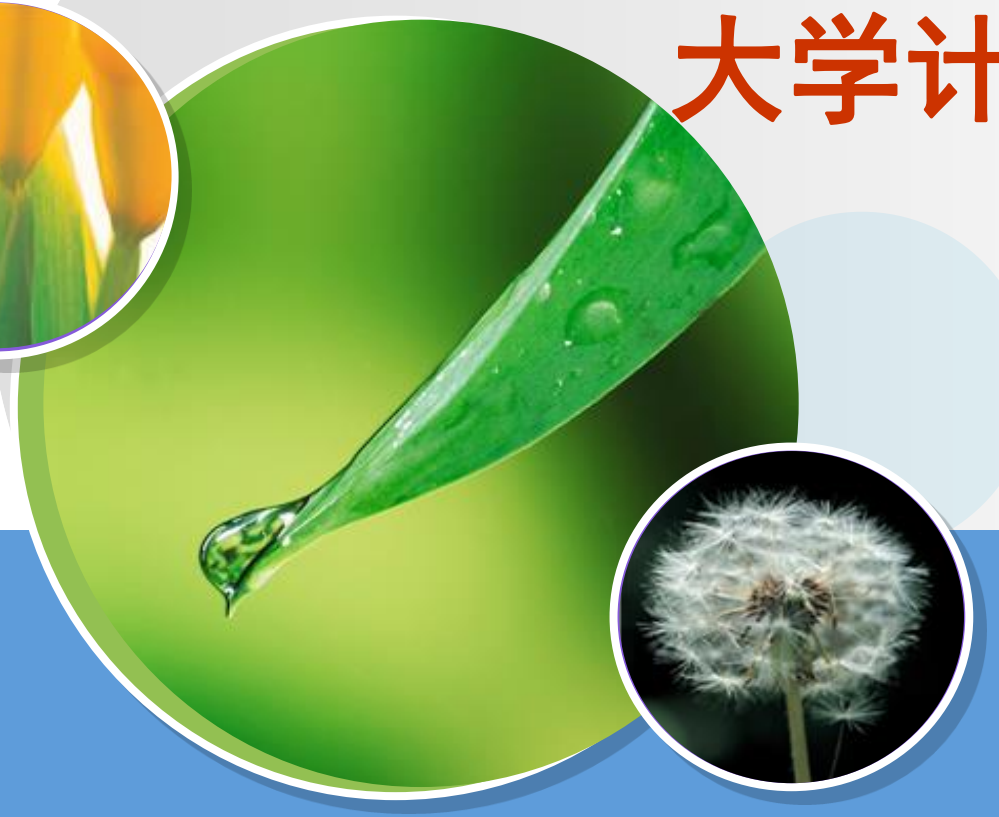
上课课件

北京航空航天大學
BEIHANG UNIVERSITY

沙河高校联盟共享课程

大学计算机基础（Python编程）



北京航空航天大学

第3章 程序设计基础与数据结构（二）

共6学时

本讲重点和难点

重点

- **分支、循环程序控制结构**
- **序列的通用操作**（索引、分片、加、乘、检查成员资格）
- **列表的各种操作**（访问、增加、删除、修改、查找元素）
- **列表的方法**（append、count、extend、index、insert、pop、remove、reverse、sort）

难点

- **序列的分片**
- **while语句**

预习任务

1、自学中国大学MOOC上北航《大学计算机基础》**MOOC**的**第5讲**视频和课件，及时完成**单元测验**和**课堂讨论**。

2、预习**教材**《面向计算思维的大学计算机基础》 **“第3章 程序设计基础与数据结构”**中**“3.3 程序控制结构 ”**、**“3.2.2 结构化数据类型简介 ”**和**“ 3.2.3 常用序列类型（sequence type） ”**。

3、预习**课件** “《大学计算机基础》-**第3章 程序设计基础与数据结构（二）【预习用】**” 。

程序控制结构

- 结构化程序设计有**3种**基本结构：**顺序控制结构**、**选择控制结构**和**迭代控制结构**
- **顺序控制结构**
  - 串行程序**按序**执行语句，当语句执行完毕则停止
- **选择控制结构（分支结构）**
  - 根据**条件**进行分支
    - ✓ if语句
- **迭代控制结构（循环结构）**
  - 在一定**条件**下重复执行**相同**的程序段
    - ✓ while语句，for语句

3.4.1 选择控制结构

- **选择结构（分支结构）：根据条件的判断确定应该执行哪一条分支的语句序列**
  - 最简单的分支控制语句为**条件语句（if语句）**
  - 条件语句结构分为三种形式
    - ✓ **非完整性**if语句
    - ✓ **二重选择**的if语句
    - ✓ **多重选择**的if语句

条件语句结构：（1）非完整性if语句

（1）非完整性if语句

当只有**一个**条件时，采用这种形式

```
if 条件表达式:
    语句序列
```

```
#if_example1.py

name=input('What is your name?')
if name.endswith('Gumby'):
    print ('Hello, Mr. Gumby')
```

其中“**条件表达式**”为**逻辑表达式或关系表达式，或布尔变量。**

```
What is your name?Jack.Gumby
Hello, Mr. Gumby
```

- **endswith**是字符串的一个方法，如果一个字符串以一个指定的**后缀**结束，则返回**True**；否则返回False

条件语句结构：（2）二重选择的if语句

（2）二重选择的if语句

当只有**一个条件**、**两个分支**时，采用这种形式

```
if 条件表达式:
        语句序列1
else:
        语句序列2
```

```
#if_example2.py

name=input('What is your name?')
if name.endswith('Gumby'):
    print( 'Hello, Mr. Gumby')
else:
    print ('Hello, stranger')
```

```
>>>
What is your name?Mingjing.Ai
Hello, stranger
```

条件语句结构：（3）多重选择的if语句

（3）多重选择的if语句

（有**两个或两个以上**条件）

```
#if_example3.py
#判断输入的数是正数、负数或零

num=float(input('Enter a number:'))

if num>0:
    print (num, 'is positive')
elif num<0:
    print (num, 'is negtive')
else:
    print (num, 'is zero')
```

```
if 条件表达式1:
        语句序列1
elif 条件表达式2:
        语句序列2
else:
        语句序列3
```

```
Enter a number:5
5.0 is positive
>>> ==============
>>>
Enter a number:-19
-19.0 is negtive
>>> ==============
>>>
Enter a number:0
0.0 is zero
```

【课堂讨论1】

- 请思考，程序改成下面这样正确吗？为什么？

```
#if_example3.py

num=float(input('Enter a number:'))

if num>0:
    print (num,'is positive')
if num<0:
    print (num,'is negtive' )
else:
    print (num,'is zero' )
```

【课堂讨论1】答案

- **不正确**
- 因为第2个elif子句“**elif num<0:**”变成了if语句“**if num<0:**”，则它与第1个if语句“**if num>0:**”是**并列**的关系，先后都会执行
- 当number为**正数**时，会产生两个输出

```
if num>0:
    print (num,'is positive')
if num<0:
    print (num,'is negtive' )
else:
    print (num,'is zero' )
```

```
Enter a number:5
5.0 is positive
5.0 is zero
>>> ===============
>>>
Enter a number:-19
-19.0 is negtive
>>> ===============
>>>
Enter a number:0
0.0 is zero
```

条件语句示例：【例3.4】电费计算

- **【例5.1】电费计算。**刘同学家今天收到了一份电费通知单，缴费标准为：月用电量小于等于150千瓦时的部分按**0.4463元/千瓦时**执行，在151~400千瓦时范围内的部分按**0.4663元/千瓦时**执行，大于400千瓦时的部分按**0.5663元/千瓦时**执行。
- 刘同学请你帮忙，编写一个Python程序，根据输入的用电总量计算应缴纳的电费是多少，从而验证电费单上的应缴电费是否正确。

- **输入**
  - ✓ 一个正整数，表示用电总量（单位以千瓦时计），不超过10000
- **输出**
  - ✓ 一个浮点数，表示应缴纳的电费，保留到**小数点后3位**（单位以元计）

【例3.4】设计思路

设计思路

1、**缴费标准分段计费，分三种情况**，则可以采用**多重选择的if语句**

- 假定用电总量为n
  - **（1）n <= 150**
  - **（2）n > 150 且 n <= 400**
  - **（3）n > 400**

2、if-elif-else结构可以写为：

```
if n <= 150:
elif n <= 400:
else:
```

说明：由于elif隐含了前一个条件”n <= 150”不成立，即n>150，故”elif n <= 400:”子句**不必写为“elif n > 150 and n <= 400:”**

【例3.4】设计思路（续）

3、分段计费

- 假定应缴纳的电费为cost

(2) 当n > 150 且 n <= 400时，分两段计算电费，再求和

小于等于150千瓦时的部分按**0.4463元/千瓦时**执行，在151~400千瓦时范围内的部分按**0.4663元/千瓦时**执行

cost = 150 * 0.4463 + (n - 150) * 0.4663

(3) 当n > 400时，分三段计算电费，再求和

cost = 150 * 0.4463 + 250 * 0.4663 + (n - 400) * 0.5663

【例3.4】的程序

例3.4-电费计算.py

```
#1、输入用电总量
n = int(input())

#2、处理（分段计算电费）
if n <= 150:
    cost = n * 0.4463
elif n <= 400:
    cost = 150 * 0.4463 + (n - 150) * 0.4663
else:
    cost = 150 * 0.4463 + 250 * 0.4663 + (n - 400) * 0.5663

#3、输出
print("%.3f" % cost)        #应缴纳的电费
```

- **程序IPO模式**
  - 用户输入（Input）
  - 处理（Process）
  - 结果输出（Output）

```
140
62.482
>>>
 RESTART: E:\amj
le\Chapter3\3.4
300
136.890
>>>
 RESTART: E:\amj
le\Chapter3\3.4
500
240.150
```

条件语句嵌套：【例3.5】

- 当某个条件成立时，如果还需要根据另一个条件是否成立来决定执行哪个语句块，可以**嵌套**使用条件语句
- 若True代码块或者False代码块**含有条件语句**
  - 称为**条件语句嵌套**

- **【例3.5】编程判断当前输入整数是否能被2，3整除。**

  - **分三种大的情况**
    - **(1) 如果能被2整除**
      - ✓ **如果**能被3整除
      - ✓ （**否则**），不能被3整除
    - **(2) （否则[不能被2整除]）如果能被3整除**
    - **(3) （否则），不能被2、3整除**

条件语句嵌套

```
if x%2 == 0:
    if x%3 == 0:
    else:
elif x%3 == 0:
else:
```

【例3.5】的程序

例3.5-divide by 2 or 3-嵌套.py

条件语句嵌套

```
x = int(input('请输入整数:'))
if x % 2 == 0:
    if x % 3 == 0:
        print('可被2整除，且被3整除')
    else:
        print('可被2整除，不能被3整除')
elif x % 3 ==0:
    print('不可被2整除，但被3整除')
else:
    print('不可被2整除，且不能被3整除')
```

```
>>>
========== RESTART:
====
请输入整数:91
不可被2整除，且不能被3整除
>>>
========== RESTART:
====
请输入整数:99
不可被2整除，但被3整除
>>>
========== RESTART:
====
请输入整数:8
可被2整除，不能被3整除
>>>
========== RESTART:
====
请输入整数:66
可被2整除，且被3整除
>>>
========== RESTART:
====
请输入整数:-33
不可被2整除，但被3整除
>>>
```

条件语句注意事项

- 条件判断
  - **绘制逻辑层次关系图**
    - ✓ 划分变量范围
    - ✓ 每个范围有相应的判断结果
  - **权衡**
    - ✓ 用多个独立的**if-elif-else**语句
    - ✓ 或**嵌套**条件语句

```
if 条件表达式1:
    执行语句1
elif 条件表达式2:
    执行语句2
else:
    执行语句3
```

```
if 条件表达式1:
    执行语句1
else:
    if 条件表达式2:
        执行语句2
    else:
        执行语句3
```

3.4.2 迭代控制结构

- **有时候，需要重复执行一些语句多次。如何编写简洁、高效的程序呢?**
- **迭代控制结构（循环结构）**：**使同一段程序执行多次的一种程序控制结构**
- 重复执行的语句序列被称为**循环体**
- 两种循环（迭代）语句
  - **for语句**
  - **while语句**

（1）for语句

- **如果需要对一个集合（序列或其他可迭代对象）的每个元素都执行同一个代码块，适合采用for语句**
  - **可迭代对象**指可以按次序迭代的对象
  - **利用for语句，可以遍历列表、元组或字典中的每个元素（键）**

```
for 变量 in 序列或其他可迭代对象:
    代码块
```

例：打印列表中每个元素

list_traversal1.py

```
name=['Alice', 'Helen', 'Peter', 'John']
print('name中的所有名字是：')
for each_item in name:
    print(each_item)
```

遍历列表

range函数

- **range函数**：Python内建的**范围函数**，**用于产生某个指定整数范围内的整数数字**
  - 产生的**整数数字**包含下限，但**不包含上限**
  - 下限为0时，可以省略

range*（下限, 上限）*

- range函数经常与**for语句**结合起来使用，指定循环迭代的范围

```
for i in range(0,10):
    print(i)
```

for语句示例：【例3.6】求累加和

【例3.6】计算1+2+3+4+…+n的累加和。

例3.6-for-累加求和.py

```
n=int(input("请输入n：  "))

sum=0                          #累加和
for i in range(1,n+1):         #i为循环变量
    sum=sum+i

print("最终累加和sum=",sum)
```

为什么是n+1?

```
>>>
请输入n：10
最终累加和sum= 55
>>> ================
>>>
请输入n：100
最终累加和sum= 5050
```

返回【课堂练习】答案

for语句示例：【例3.7】打印九九乘法表

- **【例3.7】打印如下所示的九九乘法表。**

```
1x1=1
1x2=2   2x2=4
1x3=3   2x3=6   3x3=9
1x4=4   2x4=8   3x4=12  4x4=16
1x5=5   2x5=10  3x5=15  4x5=20  5x5=25
1x6=6   2x6=12  3x6=18  4x6=24  5x6=30  6x6=36
1x7=7   2x7=14  3x7=21  4x7=28  5x7=35  6x7=42  7x7=49
1x8=8   2x8=16  3x8=24  4x8=32  5x8=40  6x8=48  7x8=56  8x8=64
1x9=9   2x9=18  3x9=27  4x9=36  5x9=45  6x9=54  7x9=63  8x9=72  9x9=81
```

- **规律**
  - 行号与**乘数**一致：**1~9**
  - 列号与**被乘数**一致：**1~9**

【例3.7】分析

	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8	j=9
i=1	1x1=1								
2	1x2=2	2x2=4							
3	1x3=3	2x3=6	3x3=9						
4	1x4=4	2x4=8	3x4=12	4x4=16					
5	1x5=5	2x5=10	3x5=15	4x5=20	5x5=25				
6	1x6=6	2x6=12	3x6=18	4x6=24	5x6=30	6x6=36			
7	1x7=7	2x7=14	3x7=21	4x7=28	5x7=35	6x7=42	7x7=49		
8	1x8=8	2x8=16	3x8=24	4x8=32	5x8=40	6x8=48	7x8=56	8x8=64	
9	1x9=9	2x9=18	3x9=27	4x9=36	5x9=45	6x9=54	7x9=63	8x9=72	9x9=81

for语句关键是要确定循环变量的范围

- **双重循环**
  - 外循环变量，**乘数**的变化：循环变量**i=1~9**，按**行**打印
  - 内循环变量，**被乘数**的变化：循环变量**j=1~i**，按**列**打印

```
for i in range(1, 10):
    for j in range(1, i+1):
```

【例3.7】程序

例3.7-for-九九乘法表.py

```
for i in range(1,10):          #i为乘数，i=1~9
    for j in range(1,i+1): #j为被乘数，j=1~i
        #（1）对于每行除最后一列外的各列
        if j<i:
            print("%dx%d=%d" % (j,i,j*i), end='\t') #采用end='\t'控制每列式子左对齐
            #end='\t'表示输出的末尾以Tab键结束
        #（2）j=i，即每行最后一列
        else:
            print("%dx%d=%d" % (j,i,j*i)) #只打印最后那个算式本身，末尾不空格
```

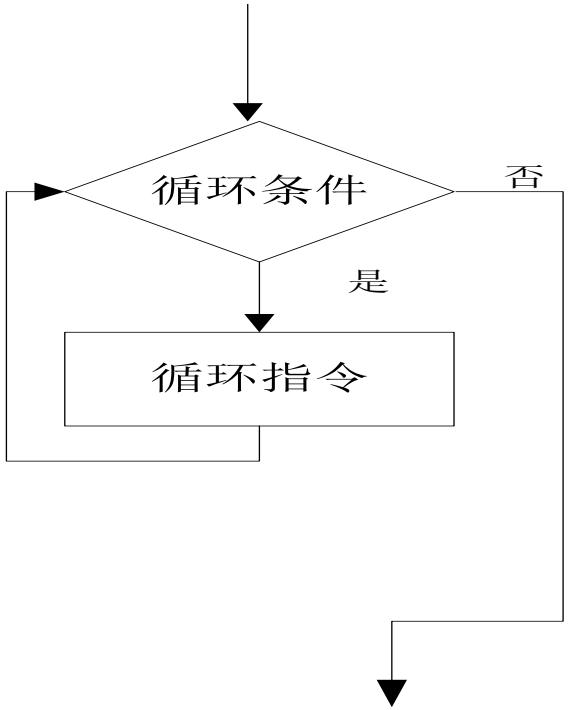
每项“**被乘数x乘数=乘积**”使用**格式化字符串方法**完成设置

- 输出**间隔**使用print()的**end**参数来控制，**end='\t'**表示输出的末尾以**Tab键**结束，则下一条print语句打印的内容将与刚才打印的内容的第一个字符**间隔8个字符输出，不换行**
- 但是， 如果所有算式都采用end= ‘\t’ 控制每列式子左对齐、不换行，每行最后一个算式的末尾会**有2~3个空格，提交到OJ上会出错**
- **解决办法**：**对于j=i，单独处理，只打印算式本身，不使用end='\t'**

- **思考**：print语句如果改写成print(“%dx%d=%d” % (i,j,i*j), end= ‘\t’)，符合题目要求吗?

（2）while语句

- **while语句**：有**条件**地执行一条或多条语句
  - **循环条件表达式**为**True**，程序将**执行循环体一次**，之后**再次评估测试**
  - 过程一直重复，**直至**测试条件评值为**False**



while 条件:
循环体

- 当**循环次数未知**时，适于使用while语句

while语句注意事项

```
while 循环条件表达式:
        语句序列
        改变循环条件表达式的值
        ……
```

1.首先判断循环条件表达式是否为真，若不为真，则其后的语句一次也不被执行！

2.在循环体中，**必须有一条改变循环条件表达式的值的语句**！或者**采用break强制退出循环**。否则，循环将无休止地进行下去

while语句示例：【例3.8】猜数字游戏

- **【例3.8】**猜数字游戏。由玩家通过键盘输入所猜数字
  - 如果玩家猜5，显示“猜对啦！”
  - 如果玩家猜测的数大于5，显示“高了！”
  - 如果猜测的数小于5，显示“低了！”

方法一 使用while语句

- 猜数情况分三类
  - 等于5
  - 小于5
  - 大于5
- 未猜中‘5’时，持续进入游戏
  - **answer**变量：1表示猜中，0表示未猜中
    - ✓ 初始化： **answer=0**
    - ✓ 进入循环的判断条件：**while(answer==0)**

方法一程序

```
print("欢迎！")
answer = 0
while (answer == 0):
    g = input("猜想数字是:")
    guess = int(g)
    if guess == 5:
        answer = 1
        print("猜对啦！")
    else:
        if guess > 5:
            print("高了！")
        else:
            print("低了！")

print("游戏结束！")
```

循环控制变量，初始化为0

循环条件判断

用户输入，并进行类型转换

更改变量answer的值——关键！

必须有此句！否则循环永远不能停止

```
>>>
========== RESTART:
====
欢迎！
猜想数字是:20
高了！
猜想数字是:-20
低了！
猜想数字是:0
低了！
猜想数字是:10
高了！
猜想数字是:5
猜对啦！
游戏结束！
```

方法一

使用while 语句

方法二 使用while True语句

- 猜数情况分三类
  - 等于5
  - 小于5
  - 大于5
- 未猜中‘5’时，持续进入游戏
  - **使用无限循环，while True:**
- 猜中‘5’时，使用**break**
  - 跳出本重循环

方法二程序

```
print("欢迎！")
while True:
    g = input("猜想数字是:")
    guess = int(g)
    if guess == 5:
        print("猜对啦！")
        break
    else:
        if guess > 5:
            print("高了！")
        else:
            print("低了！")

print("游戏结束！")
```

持续进入循环

用户输入，并类型转换

打印输出相应内容

跳出本重循环——关键！

```
====================== RESTART:
===
欢迎！
猜想数字是:21
高了！
猜想数字是:-21
低了！
猜想数字是:0
低了！
猜想数字是:11
高了！
猜想数字是:6
高了！
猜想数字是:3
低了！
猜想数字是:5
猜对啦！
游戏结束！
```

方法二

使用while True语句

方法一和方法二程序比较

```
print("欢迎！")
answer = 0
while (answer == 0):
    g = input("猜想数字是:")
    guess = int(g)
    if guess == 5:
        answer = 1
        print("猜对啦！")
    else:
        if guess > 5:
            print("高了！")
        else:
            print("低了！")

print("游戏结束！")
```

方法一

```
print("欢迎！")
while True:
    g = input("猜想数字是:")
    guess = int(g)
    if guess == 5:
        print("猜对啦！")
        break
    else:
        if guess > 5:
            print("高了！")
        else:
            print("低了！")

print("游戏结束！")
```

方法一

使用while True语句更简洁

【课后练习】

【课后练习】由用户输入一个整数n，采用**while**语句，计算1+2+3+4+…+n的累加和。写出相应程序。

- 与采用for语句相比，哪种方法程序更简洁?

循环控制语句：continue和break语句

1、continue语句

- **终止当前循环**，并忽略continue后面的语句；回到循环顶端，**提前进入下一轮循环**

2、break语句

- 在while循环和for循环中均可使用
- 一般放在**if选择结构**中，当满足特定条件时，一旦遇到break语句，则立即跳出循环，使得**整个循环提前结束，继续执行循环结构后面的语句**

> - 除非**break语句**让代码更简单或更清晰，否则不要轻易使用

while True语句示例【例3.9】

【例3.9】 输入正整数n，求1~n之间的全部奇数之和。

- **设计思路**
  - 设x是1~n之间的任意一个整数，初值为0
  - 采用**while True**循环语句
    - ✓ x逐次加1
    - ✓ （1）如果x是偶数，直接进行下一轮循环（**continue**语句）；
    - ✓ （2）如果x大于n，终止循环（**break**语句）；
    - ✓ （3）如果x是奇数，则对x累加求和

【例3.9】程序

```
#例3.9-while True示例-求1~n的全部奇数之和.py

n = int(input("输入任意一个正整数："))  #输入任意一个正整数

x=0                 #1~n之间的任意一个整数，初值为0
ans=0               #累加求和结果
while True:         #无限循环
    x += 1          #x逐次加1
    if x%2 == 0:    #（1）如果x是偶数，直接进行下一轮循环
        continue
    elif x > n:     #（2）如果大于n，终止循环
        break       跳出循环
    else:           #（3）如果是奇数，则累加求和
        ans += x

print("ans=",ans)
```

```
输入任意一个正整数： 10
ans= 25
>>>
 RESTART: E:\amj\course\Compu
语法\5.1  程序控制结构\while\
输入任意一个正整数： 100
ans= 2500
>>>
```

3.5 结构化数据类型

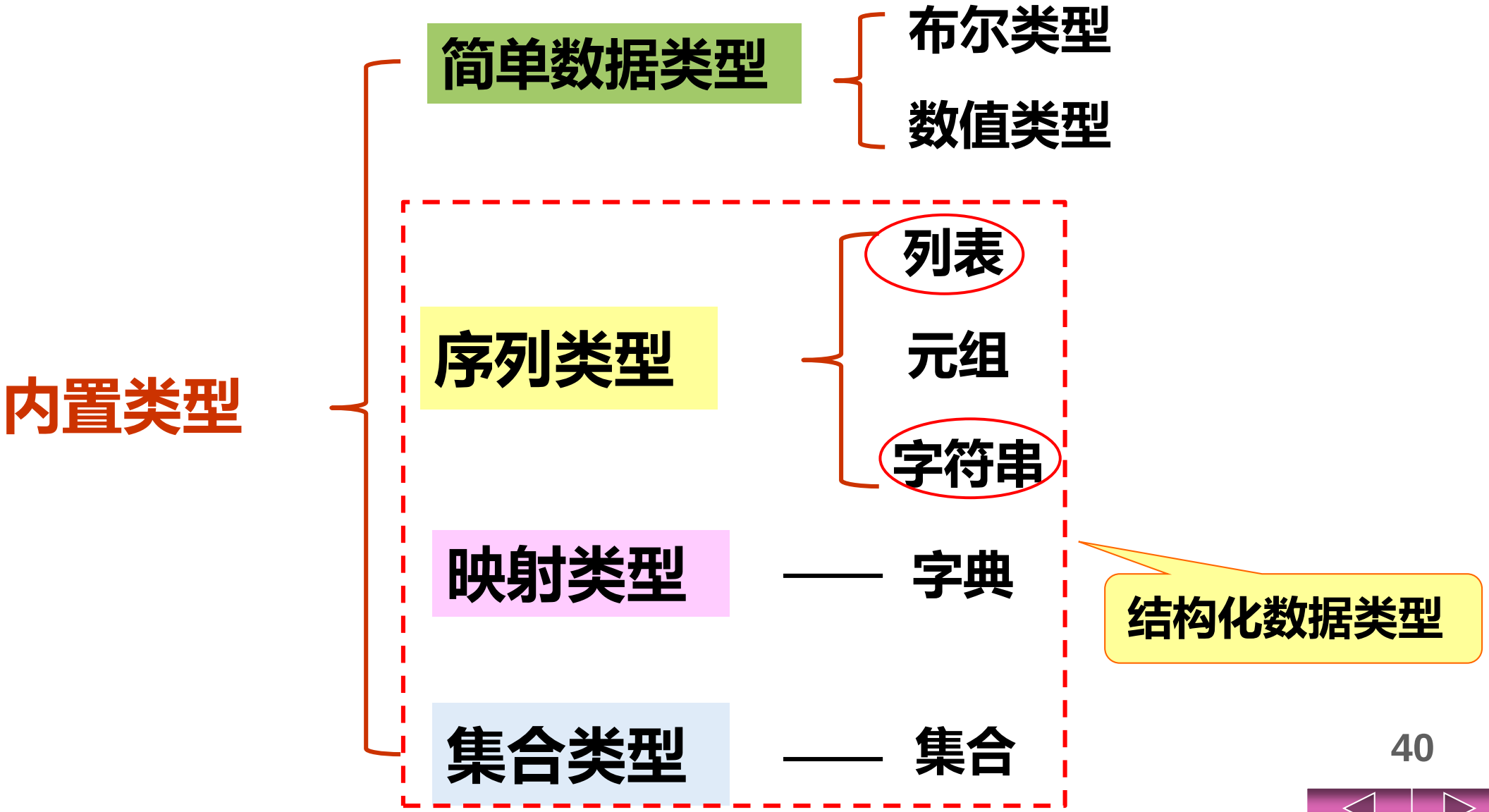
- 3.5.1 概述
- 3.5.2 列表
- 3.5.3 字符串
- 3.5.4 字典

3.5.1 概述

问题：少量数据可以用单独的变量名存储，批量数据如何存储?

- **Python常用内置类型**：简单数据类型、序列类型、映射类型和集合类型
  - **简单数据类型**包括**布尔类型**和**数值类型**，**没有内部结构**
- **序列类型、映射类型和集合类型属于结构化数据类型**
  - 数据内部由若干分量组成（数据有“**内部结构**”）
  - 数据之间可能存在特定的**逻辑关系**
- Python中，批量数据可以采用**结构化数据类型**来存储

常用的Python内置类型



数据结构

- **数据结构**：通过某种方式（如对元素进行编号）组织在一起的、具有**相同属性**的数据元素（数字或字符）的**集合**
- Python的数据结构：**序列**（sequence），**映射**（mapping），**集合**（set）
  - **序列**：**由整数索引的对象的有序集合**。数据成员是有序排列的，可以通过元素的位置访问一个或多个成员元素
  - **索引**：**序列中的每个元素被分配一个序号，即元素的位置**。第一个索引是0，第二个索引是1，依此类推

索引：　0　　1　　2　　3　　4

- 例如列表：[‘a’ , ‘b’ , ‘c’ , ‘d’ , ’ e’]

-5　　-4　　-3　　-2　　-1

序列的通用操作

序列

由整数索引的对象的有序集合

- **序列的通用操作**
  - **索引**（indexing）：通过元素编号**访问（获取）**序列中的某个元素
  - **分片**（slicing）：访问序列中的**一定范围（间隔一定步长）内**的元素
  - **加**（adding）：使用加号**连接**两个或两个以上的序列成为一个**新**序列
  - **乘**（multiplying）：将原序列**重复**若干次，连接成一个**新**序列
  - **成员检测**：使用**成员检测运算符in**检查一个值是否在某个序列中
- **内建函数**：计算序列**长度** len(x)、找出最大元素max(x)和最小元素min(x)、求和函数sum(x)

详见教材和MOOC

（1）索引（indexing）

- **索引**：通过元素编号访问（获取）序列中的某个元素
  - 序列名后跟一对**方括号**，将要访问的元素编号括起来。
  - **正数索引**：**最左边**的元素编号为**0**，从左到右依次为0、1、2、……
  - **负数索引**：从**最右边**（即最后1个元素）开始计数，即最后1个元素的编号为**-1**，倒数第二个元素的编号为-2，……

```
>>> greeting='Hello'
>>> greeting[0]
'H'
>>> greeting[-5]
'H'
```

访问字符串中的最左侧元素

负数索引：访问字符串中的从右至左的第5个元素

（2）分片（slicing）

- **分片**：访问序列中的一定范围内的元素
  - 提取序列的一部分

步长为1时可以省略

格式：**<列表名>[索引1:索引2:步长]**

```
>>> greeting = 'Hey, man!'
>>> greeting[5:8]
'man'
```

greeting[5:7]，则只能提取‘ma’

  - **在普通的分片中，默认步长为1（一般是隐式设置），**返回**索引1**和**（索引2-1）**之间的所有元素

注意：返回的元素不包括第2个索引对应的元素！

副本

```
>>> numbers = [1,4,9]
>>> numbers[:]
[1, 4, 9]
>>> y = numbers[:]
>>> y
[1, 4, 9]
>>> y.pop()
9
>>> y
[1, 4]
>>> numbers
[1, 4, 9]
```

希望访问整个列表，则将两个索引都置空

赋给另一个变量，生成**副本**

改变副本

不会影响原列表

- **提示**：通过**将两个索引都置空，**可以方便地**产生一个原列表的副本**。这时如果对原列表的**副本**进行任何**操作**（如修改元素、删除某元素、排序），**不会影响到原列表**

（3）加（adding）

- **序列相加**
  - **连接**两个或两个以上的序列成为一个**新**序列

原序列不变

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> 'Hello, ' + 'world!'
'Hello, world!'
>>> [1, 2, 3] + 'world!'
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    [1, 2, 3] + 'world!'
TypeError: can only concatenate list (not "str") to list
```

相同类型

只能将列表与列表或者字符串与字符串进行adding操作

无法将字符串add至列表

- **注意**：**必须是相同类型的序列才能进行连接操作**。列表和字符串是无法连接在一起的！

（4）乘（multiplying）

- **序列乘法**：用数字x乘以一个序列
  - 将原序列**重复x次**，生成一个新的序列

```
>>> 'hello' * 5
'hellohellohellohellohello'
>>> [37, 38, 39] * 4
[37, 38, 39, 37, 38, 39, 37, 38, 39, 37, 38, 39]
```

- **注意**：**并不是将序列中每个元素乘以x！**

（5）检查成员资格（使用in）

- **检查成员资格**：**使用in运算符检查一个值是否在某个序列中**
  - 如果在，则返回**True**
  - 如果不在，则返回**False**
- **应用**：for语句对列表、字符串、字典的遍历

in_遍历序列.py

```
name=['Alice', 'Helen', 'Peter', 'John']
print('name中的所有名字是：')
for each_item in name:
    print(each_item)

word='hardwork'
print('word中的所有字母是：')
for each_item in word:
    print(each_item)
```

```
name中的所有名字是：
Alice
Helen
Peter
John
word中的所有字母是：
h
a
r
d
w
o
r
k
```

3.5.2 列表

- **列表**（List）：值的**有序**序列，每个值由索引来识别
  - 用一对**方括号**包裹多个元素，各个元素通过**逗号**分隔
  - 如： [1, 10, 100, 1000]

    ['Alice', 'Helen', 'Peter', 'John']
- Python中最具灵活性的**有序集合对象**类型

[0, 1, 2, 3, 4, 5, 6]

[0.0, 1.1, 2.2, 3.3]

[‘a’, ‘b’, ‘c’, ‘d’, ’e’]

[‘a’, 1.1, 2, ‘d’, 3.3]

[[0,1], [1.1, 3], ’c’]

[‘北京’, ‘上海’, ‘广州’]

1、列表的主要性质

1、列表的主要性质

- **可变长度**、**异构**以及任意**嵌套**
  - **长度可变**：列表可以根据需要增长或缩短
  - **异构**：可以包含任何类型的对象（**数字**、**字符串**、**自定义对象**甚至**其他列表**）
  - 支持任意类型对象的**嵌套**
- **可变**序列
  - 列表支持在**原处**的修改，通过**列表方法**（**remove**、 **pop**）调用、**删除语句**等方法
- 响应针对**序列**的**通用操作**：**索引、分片、连接**和**乘法、成员资格检查**

2、列表的创建和访问

- **列表的创建**

0 1 2 3 4 5

```
a_list = [ 'a', 'b', 0, 'z', 2020, 'example' ]
```

-6 -5 -4 -3 -2 -1

异构

  - 各个元素都有序号（**索引**），两种索引方式：**正数索引**，**负数索引**
  - **列表的创建，通过[ ]赋予列表对应的值**
  - **[ ]**表示空列表，如：n_list = [ ]
  - 列表可以**嵌套**，即构建多维的列表

二维列表：a_list = [[10, 20, 30]， [40, 50, 60]，[70, 80, 90]]

  - 创建一个所有初始值都为0的列表

```
>>>lis=[0 for i in range(10)]
>>>lis
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

访问列表

- 示例：

```
          0    1    2    3      4           5
a_list = [ 'a', 'b', 0, 'z', 2020, 'example' ]
          -6   -5   -4   -3     -2          -1
```

- **列表按照索引序号访问一个元素**
  - a_list[0] == a_list[-6]
  - a_list[-1] == a_list[5]
  - a_list[1] == a_list[?]
  - ***如果索引越界会怎么样?*** 比如a_list[6]和a_list[-7]

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    a_list[6]
IndexError: list index out of range
```

切片访问多个元素

- 示例：

```
          0    1    2    3       4            5
a_list = [ 'a', 'b', 0, 'z', 2020, 'example' ]
          -6   -5   -4   -3      -2           -1
```

- **切片访问多个元素**
  - 通过指定两个索引值，可以从列表中获取称作“**切片**”的某个部分，返回值是一个**新**列表，按顺序从第一个切片索引开始，到第二个切片索引截止但**不包含第二个切片索引**

    a_list[1:3] == ['b', 0]

    不包含a_list[3]

    思考：a_list[0:4:2]=？
  - 如果左切片索引为**零**，可以将其**留空**而将零隐去

    a_list[:3] 与 a_list[0:3] 相同
  - 如果右切片索引为列表的**长度**，也可以将其**留空**

    a_list[3:] 与 a_list[3:6] 相同，因为该列表有六个元素

列表的遍历

- **列表的遍历**　要掌握！

遍历：逐一访问列表中的每个元素

- 使用for语句遍历

```
# list_traversal1.py
name=['Alice', 'Helen', 'Peter', 'John']
print('name中的所有名字是：')
for i in name:
    print(i)
```

循环变量为列表中每个元素

```
name中的所有名字是：
Alice
Helen
Peter
John
>>>
```

3、列表的主要方法

- **列表方法**

要掌握！

  - **方法**：是针对对象属性的各种**操作**。是能执行特定功能的程序语句块，即**函数**
  - **方法的调用**：**对象.方法(参数)**
  - Python为列表定义了多个方法，用于**检查**或**修改**列表中的内容
    - ✓ append
    - ✓ count
    - ✓ extend
    - ✓ index
    - ✓ insert
    - ✓ pop
    - ✓ remove
    - ✓ reverse
    - ✓ sort

列表方法的使用

方法名称	含义	示例	说明

append	在列表末尾追加**一个**新的对象
	.append(<追加对象>)
		>>> prices=[1, 2, 3]
		>>> prices.append(4)
		>>> prices
		[1, 2, 3, 4]
			在恰当位置**修改**原来的**列表**（即列表名不变），而不是返回一个修改过的新列表。
			一次只能追加**一个**对象

count	统计某个元素在列表中出现的次数 **.count(<某元素>)**	>>> ['to', 'be', 'or', 'not', 'to', 'be'].count('to') 2	

extend	在列表的末尾一次性追加另一个序列中的**多个**值
	.extend(<另一个序列>)
		>>> x=[1, 2, 3]
		>>> y= 'abc'
		>>> x.extend(y)　　#在x列表的末尾一次性追加字符串y中的各字符
		>>> x
		[1, 2, 3, 'a', 'b', 'c']
			与连接操作有区别。extend**修改**了原来的**列表**，连接操作则不修改原始序列，而返回一个全新的序列。

■ **思考：append（附加）与extend（扩展）有何区别?**

列表方法的使用（续1）

方法名称	含义	示例	说明

index	从列表中找出某个值的第一个匹配项的索引位置
	.index(<要匹配的值>)
		>>> greeting=['Good', 'morning', 'everyone', 'she', 'said']
		>>> greeting.index('everyone')
		2
			可用于**查找**某个**元素**在列表中的**位置**

insert	将**一个**对象插入列表中指定位置
	.insert(<索引>,<待插入对象>)
		>>> numbers=[1, 2, 3, 5, 6, 7]
		>>>numbers.insert(3, 'four')
		#在第3个编号位置插入 “four”
		>>> numbers
		[1, 2, 3, 'four', 5, 6, 7]
			一次只能插入一个对象。可以使用**分片赋值**实现插入**多个对象**的操作

```
>>> numbers=[1, 2, 3, 8, 9, 10]
>>> numbers[3:3]= [4,5,6,7]      #在第3个元素之前，插入新的元素
>>> numbers
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

使用分片赋值
实现插入操作

列表方法的使用（续2）

方法名称	含义	示例	说明

pop	移除列表中索引所指的一个元素（**默认为最后一个**），并返回该元素的值
	.pop(<索引>)
		>>> x=[4, 5, 6, 7]
		>>> x.pop(1)
		5
		
		>>> x.pop()
		#括号中不写参数则移除列表中**最后一个**元素
		7
			pop方法是唯一一个既能**修改原列表**又能**返回元素值**（除了None）的列表方法

remove	移除某个值在列表中的**第一个**匹配项
	.remove(<要移除的元素>)
		>>> x=['to‘ , 'be', 'or', 'not', 'to', 'be']
		>>> x.remove('be')
		>>> x
		['to', 'or', 'not', 'to', 'be']
			如果列表中有多项都与remove要移除的值相同，也只移除**第一个**匹配项。
			与pop方法不同，remove方法**修改原列表**，**但不返回值**

■ **思考：** **pop与remove有何区别?**

【课堂讨论2】

- 已知s1=[1,2,3,4,5]，s2=[]， s3=[] 。
  - （1）执行**s2.append(s1.pop(0))**后，**s1=? s2=?**
  - （2）如果希望将s1中的元素“5”添加到s3中，

    s3.append(s1.remove(5))正确吗？为什么？**s3=?**
  - （3）将s1中的元素“5”添加到s3中的正确写法是什么？

【课堂讨论2】答案

- 已知s1=[1,2,3,4,5]，s2=[]， s3=[]
  - **（1）s2.append(s1.pop(0))** #弹出s1中第0个元素，添加到s2末尾

    s2=[1]

    s1=[2, 3, 4, 5]
  - **（2）s3.append(s1.remove(5))不正确。**

    因为remove方法**移除**某个值在列表中的**第一个匹配项**，但在调用该方法的地方并**不返回**这个**值**。所以s1.remove(5)不能获得“5”，也就无法添加到s3中。

    s3=[None]
  - **（3）将s1中的元素“5”添加到s3中的正确写法**

    s3.append(s1.pop()) #弹出s1中最后一个元素，添加到s3末尾

列表方法的使用（续3）

方法名称	含义	示例	说明

reverse	将列表中的元素**反向**存放
	.reverse()
		>>> x=[4, 5, 9, 8]
		>>> **x.reverse()**
		>>> x
		[8, 9, 5, 4]
			该方法**修改原列表**，但**不返回值**

sort	在原位置对列表进行**排序** **.sort()**按从小到大顺序排序， 或**.sort(key=<参数>)** 或**.sort(reverse=<参数>)** 指明列表是否**反向排序（从大到小）**	>>>x = [4,6,2,1,7,9] >> x.sort() >>> x [1, 2, 4, 6, 7, 9] >>>y= [4,6,2,1,7,9] >>>**y.sort(reverse =True)** >>>y [9,7,6,4,2,1]	该方法**修改原列表**，让其中的元素能按一定的顺序排列。但**不返回值**

思考：列表的sort方法与sorted函数的区别?

关于sort方法

- 如果列表元素是**字符串**，sort方法也可以对其排序
- 如果每个字符串包含不止一个字符，先按最左边字符的ASCII码值大小排序；当最左边的字符相同时，再按第2个字符的ASCII码值大小排序

```
# sort方法-字符串排序.py
fruit=['pear', 'banana', 'peach', 'apple', 'strawberry', 'pineapple']
fruit.sort()                          #按ASCII码值从小到大顺序排序
#fruit.sort(reverse =True)            #按ASCII码值从大到小顺序排序

print(fruit)
```

```
>>>
['apple', 'banana', 'peach', 'pear', 'pineapple', 'strawberry']
```

4、列表的各种操作

- **列表的各种操作**
  - **增加**元素
  - **删除**元素
  - **修改**元素
  - **查找**元素
  - 列表元素**排序**
  - 与其他数据类型的**转换**
  - 其他**内建函数**

列表的操作（1）：增加元素

- 四种方法可以给列表**增加新的元素**
  - **方法一**：+连接运算符：将别的列表元素添加到本列表尾部，产生一个**新**的列表（**不改变原列表**）

```
>>> list3=[1,2,3,'a','b']
>>> list3+['add1', 'add2']
[1, 2, 3, 'a', 'b', 'add1', 'add2']
>>> list3
[1, 2, 3, 'a', 'b']
```

  - **方法二**：append(x)方法，将括号中的**对象元素**添加到列表末尾
  - **方法三**：extend(L)方法，将括号中**序列**的**各个元素**追加到列表末尾
  - **方法四**：insert(i, x)方法，将元素x添加到索引i指定位置

- **注意**：**方法二~方法四均修改了原列表！**

append()方法：在末尾添加一个对象

- **append(x)方法**，将括号中的对象元素（只有**一个**）添加到列表末尾，相当于**a[len(a):] = [x]**
- **示例**
  - list4=[1, 2, 3, 'add1', 'add2']
  - list4.append('app1')　　这时候list4里面是什么?
    [1, 2, 3, 'add1', 'add2', 'app1']
  - list4.append('app1', 'app2')　为什么错误? ——**只能有一个参数**
  - list4.append(['app3', 'app4']) ——list4里面又是什么?
    [1, 2, 3, 'add1', 'add2', ['app3', 'app4']]

说明:

- append(x)添加一个元素到列表中，['app1', 'app2']则为一个新的元素，不过这个元素是一个**列表**而已

extend()方法在末尾添加多个对象

- **extend(L)方法**，将序列中的**各个**元素逐个**追加**到列表末尾
- 通过添加指定列表的所有元素来扩充列表，相当于**a[len(a):]=L**
- 示例：
  - list4 =[1, 2, 3, 'add1', 'add2']
  - list4.extend('app1')　　这时候list4里面是什么？

修改了原列表

与list4.apptend ('app1') 相同吗?

[1, 2, 3, 'add1', 'add2', 'a', 'p', 'p', '1']

  - list4.extend([0,1,2])　　list4里面又是什么？

[1, 2, 3, 'add1', 'add2', 0, 1, 2]

  - list4.extend('app1', 'app2') 为什么错误？——因为**只能有一个参数**

```
TypeError: extend() takes exactly one argument (2 given)
```

- **说明**：extend(L)将序列中所有元素**逐个**添加到原列表中

列表的操作（2）：删除元素

- 四种途径**删除元素**

方法一

- 使用**del语句**删除列表中的**某个**或**连续几个**元素
- **注意**：del语句还可以删除其他元素，如字典元素

```
>>> names=['Alice', 'Helen', 'Peter', 'John']
>>> del names[1]      #删除某个元素
>>> names
['Alice', 'Peter', 'John']

>>> del names[0:2]    #删除连续的几个元素
>>> names
['John']

>>> lst = [1, 2, 3]
>>> del lst[0:]       #删除所有元素
>>> lst
[]
```

空列表

- **说明**：如果列表不再需要使用，可以使用del语句删除整个列表

  del 列表名

列表的操作（2）：删除元素（续1）

方法二

- 使用**pop方法**删除**指定位置**元素并弹出
  - ✓ pop (i)　删除第i个元素并弹出（这里i为索引）
  - ✓ pop ()　删除最后一个元素并弹出

- **示例**:

```
>>> lis=[1,2,3, 'test', 'pop1', 'pop2']
>>> lis.pop(2)     #弹出索引为2的元素，返回值为该元素
3
>>> lis
[1, 2, 'test', 'pop1', 'pop2']

>>> lis.pop()      #弹出最后一个元素
'pop2'
>>> lis
[1, 2, 'test', 'pop1']
```

列表的操作（2）：删除元素（续2）

方法三

- 按照**元素值**删除： **remove (x)**，移除某个值在列表中的**第一个**匹配项

```
>>> lis=[1,2,3, 'test', 'pop1', 'pop2', 'test' ]
>>> lis.remove (‘test’)                    #删除第一个 'test'
>>> lis
[1, 2, 3, 'pop1', 'pop2‘, 'test']
```

第二个匹配项

方法四

- 删除列表中所有元素： **clear ()**

```
>>> lis=[1,2,3, 'test', 'pop1', 'pop2', 'test' ]
>>> lis.clear ()
>>> lis
[]
```

列表的操作（3）：列表元素排序

两种方法进行**列表元素排序**

- 假定已创建列表lis
- **方法一**：**lis.sort()**　　对列表中的元素按一定规则排序。**原列表被修改**
- **方法二**：**sorted(lis)**　　对列表中的元素临时排序，返回副本；**原列表不变**

```
# sorted函数-字符串排序.py
fruit=['pear', 'banana', 'peach', 'apple', 'strawberry', 'pineapple']
new_lis=sorted(fruit)    #按ASCII码值从小到大顺序排序

print('排序后副本：', new_lis)
print('原列表不变：', fruit)
```

```
>>>
排序后副本：  ['apple', 'banana', 'peach', 'pear', 'pineapple', 'strawberry']
原列表不变：  ['pear', 'banana', 'peach', 'apple', 'strawberry', 'pineapple']
```

【课堂讨论3】

- 已知s1=[1, 2, 3, 4] 。
  - （1）执行**s2=s1，s2.reverse()**后，**s2=？ s1=?**
  - （2）如何使**s1**保持**不变**？给出**源代码**

三分钟内完成

【课堂讨论3】答案

- **（1）执行s2=s1，s2.reverse()后 s2、s1均为[4, 3, 2, 1]**

  因为s2=s1是使s2与s1指代**同一对象**[1, 2, 3, 4] 。如果修改s2，则s1会随之变化。

```
>>> s1=[1, 2, 3, 4]
>>> s2=s1
>>> s2.reverse()
>>> s2
[4, 3, 2, 1]
>>> s1
[4, 3, 2, 1]
```

- **（2）如何使s1保持不变?**

```
s1=[1, 2, 3, 4]
s2=s1[:]
s2.reverse()
```

s2是s1的一个副本

```
>>> s1=[1, 2, 3, 4]
>>> s2=s1[:]
>>> s2.reverse()
>>> s2
[4, 3, 2, 1]
>>> s1
[1, 2, 3, 4]
```

技巧：sort方法如何保持原列表不变？

- **sort方法**、**reverse方法**用于在原位置对列表进行排序或反向存放，这意味着经过操作，**原列表**被**改变**了
- **但如果用户需要一个排好序或反向存放的列表副本，同时又保留原列表不变，怎么做?**

- **方法一**：先采用分片（调用**x[:]**）**复制整个列表**给另一个变量（y），再对该变量（y）排序，则原列表不变

不能写为y=x

```
x=[4,6,2,1,7,9]          #原始数据

y=x[:]                   #获取x的一个副本，以免修改了x本身
y.sort()                 #对副本进行排序

print('排序后的y: ',y)
print('此时列表x: ',x)
```

```
排序后的y:  [1, 2, 4, 6, 7, 9]
此时列表x:  [4, 6, 2, 1, 7, 9]
```

copy模块的deepcopy函数

- **方法二：调用copy模块的deepcopy函数进行深拷贝**
  - 修改s2，s1并没有随之变化

```
import copy

s1=[1,2,3,7,6]
s2=copy.deepcopy(s1)
s2.reverse()

print('反向后的s2=',s2)
print('s1=',s1)
```

```
反向后的s2= [6, 7, 3, 2, 1]
s1= [1, 2, 3, 7, 6]
```

- **或者调用copy模块的copy函数（浅拷贝）**

```
import copy

s1=[1,2,3,7,6]
s3=copy.copy(s1) #产生s1的一个副本
s3.reverse()

print('反向后的s3=',s3)
print('s1=',s1)
```

```
反向后的s3= [6, 7, 3, 2, 1]
s1= [1, 2, 3, 7, 6]
```

关于深拷贝和浅拷贝

- **比较y=x[:]、 copy模块的deepcopy函数、copy模块的copy函数的异同**
  - **deepcopy函数**：**深拷贝（deep copy，深复制）**，将被复制对象完全再复制一遍，作为独立的新个体单独存在。修改原有被复制对象不会对副本产生影响；对副本的修改也不会影响原对象
  - **copy函数**：**浅拷贝（shallow copy）**，仅复制对象本身，而不对其中的子对象进行复制。如果对**原子对象**进行修改，则副本中的**子对象**也会随之修改；反之亦然。**y= x[:]**也是**浅拷贝**
    - ✓ 对于**简单**的对象，copy 与 deepcopy **没有区别，**修改原对象不会对副本产生影响
    - ✓ 对于**复杂**对象（序列里的嵌套序列，字典里的嵌套序列），二者则**不同**

- **深层复制 deepcopy()：复制了对象和对象的所有子对象**
- **浅层复制 copy()：复制父对象，子对象仍然使用引用的方式**

深拷贝与浅拷贝的比较

```
>>> import copy
>>> a=[1,2,3]
>>> b=copy.copy(a)
>>> c=copy.deepcopy(a)
>>> b==c
True
>>> a[2]=9
>>> b
[1, 2, 3]
>>> c
[1, 2, 3]
>>> a
[1, 2, 9]
```

对于**简单**的对象，copy 与 deepcopy **没有区别**，修改原对象不会对副本产生影响

```
>>> a=[1,2,['asd',6]]
>>> b=copy.copy(a)
>>> c=copy.deepcopy(a)
>>> d=a[:]
>>> a[2][0]='1'
>>> a
[1, 2, ['1', 6]]
>>> b
[1, 2, ['1', 6]]
>>> c
[1, 2, ['asd', 6]]
>>> d
[1, 2, ['1', 6]]
>>>
```

对于**复杂**的对象，修改**原子对象**，copy 的**副本**随之**修改**

对于复杂的对象，修改**原子对象**，deepcopy 的**副本不受影响**

copy-deepcopy比较.py

转换为列表及列表统计

- **内建函数**
  - **list**(seq)：将其他序列类型（如字符串）**转换为列表**

```
>>> list('abcd')
['a', 'b', 'c', 'd']
```

  - **len()**：取得列表元素个数

```
>>> s1=[10, 20, 30, 40, 50, 60, 70]
>>> len(s1)
7
```

  - **sum()**：列表求和

```
>>> sum(s1)
280
```

自行练习

  - **max()**：求最大值

```
>>> max(s1)
70
```

  - **min()**：求最小值

```
>>> min(s1)
10
```

  - **count() 方法**：返回某个元素在列表中出现的次数

```
>>> ['to', 'be', 'or', 'not', 'to', 'be'].count('to')
2
```