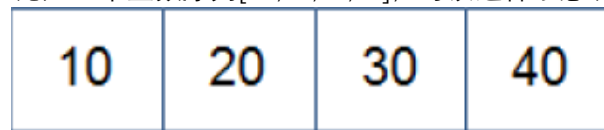


# 序列与控制语句

整理 by Orash  
For 2020 Summer Session

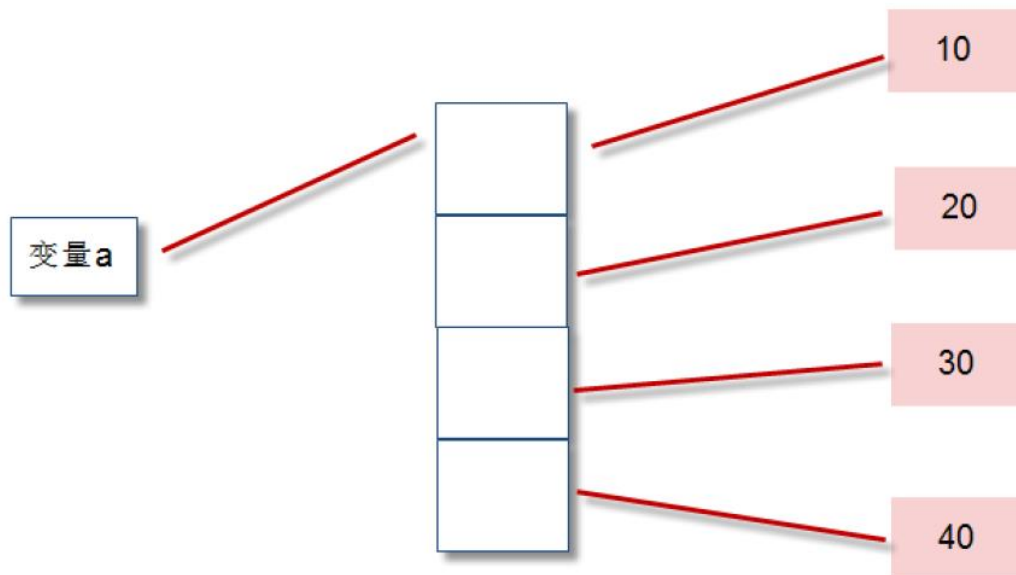
# 序列

序列是一种数据存储方式，用来存储一系列的数据。在内存中，序列就是一块用来存放多个值的连续的内存空间。比如一个整数序列[10,20,30,40]，可以这样示意表示：



由于Python3中一切皆对象，在内存中实际是按照如下方式存储的：

`a = [10,20,30,40]`



从图示中，我们可以看出序列中存储的是整数对象的地址，而不是整数对象的值。python中常用的序列结构有：

字符串、列表、元组、字典、集合

我们上一节课学习的字符串就是一种序列。关于字符串里面很多操作，在这一章中仍然会用到，大家一定会感觉非常熟悉。

本章内容，我们必须非常熟悉。无论是在学习还是工作中，序列都是每天都会用到的技术，可以非常方便的帮助我们进行数据存储的操作。

### 3.1列表

#### 3.1.1 列表简介

**列表：用于存储任意数目、任意类型的数据集合。**

列表是内置可变序列，是包含多个元素的有序连续的内存空间。列表定义的标准语法格式：

```
a = [10,20,30,40]
```

其中，10,20,30,40这些称为：列表a的元素。

列表中的元素可以各不相同，可以是任意类型。比如：

```
a = [10,20,'abc',True]
```

列表对象的常用方法汇总如下，方便大家学习和查阅。

方法	要点	描述
list.append(x)	增加元素	将元素 x 增加到列表 list 尾部
list.extend(aList)	增加元素	将列表 alist 所有元素加到列表 list 尾部
list.insert(index,x)	增加元素	在列表 list 指定位置 index 处插入元素 x
list.remove(x)	删除元素	在列表 list 中删除首次出现的指定元素 x
list.pop([index])	删除元素	删除并返回列表 list 指定为止 index 处的元素，默认是最后一个元素
list.clear()	删除所有元素	删除列表所有元素，并不是删除列表对象
list.index(x)	访问元素	返回第一个 x 的索引位置，若不存在 x 元素抛出异常
list.count(x)	计数	返回指定元素 x 在列表 list 中出现的次数
len(list)	列表长度	返回列表中包含元素的个数
list.reverse()	翻转列表	所有元素原地翻转
list.sort()	排序	所有元素原地排序
list.copy()	浅拷贝	返回列表对象的浅拷贝

Python的列表大小可变，根据需要随时增加或缩小。

字符串和列表都是序列类型，一个字符串是一个字符序列，一个列表是任何元素的序列。我们前面学习的很多字符串的方法，在列表中也有类似的用法，几乎一模一样。

### 3.1.2 列表的创建

#### 基本语法[]创建

```
>>> a=[10, 20, 'sx', 'hsj', 'zyq']
>>> a
[10, 20, 'sx', 'hsj', 'zyq']
>>> a=[]
>>> a
[]
```

#### list()创建

使用list()可以将任何可迭代的数据转化成列表。

```
>>> a=list()
>>> a=list(range(10))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a=list("I love JDFZ!")
>>> a
['I', ' ', 'l', 'o', 'v', 'e', ' ', 'J', 'D', 'F', 'Z', '!']
>>> |
```

#### range()创建整数列表

range()可以帮助我们非常方便的创建整数列表，这在开发中及其有用。语法格式为：

range([start,] end [,step])

start参数：可选，表示起始数字。默认是0

end参数：必选，表示结尾数字。

step参数：可选，表示步长，默认为1

python3中range()返回的是一个range对象，而不是列表。我们需要通过list()方法将其转换成列表对象。

典型示例如下：

```
>>> list(range(3, 15, 2))
[3, 5, 7, 9, 11, 13]
>>> list(range(15, 3, -1))
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4]
>>> list(range(3, -10, -1))
[3, 2, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(3, -10, 1))
[]
```

#### 推导式生成列表(简介一下，重点在 for 循环后讲)

使用列表推导式可以非常方便的创建列表，在开发中经常使用。但是，由于涉及到 for 循环和 if 语句。在此，仅做基本介绍。在我们控制语句后面，会详细讲解更多列表推导式的细节。

```

>>> a=[x*2 for x in range(5)]
>>> a
[0, 2, 4, 6, 8]
>>> a=[x*2 for x in range(100) if x%9==0]
>>> a
[0, 18, 36, 54, 72, 90, 108, 126, 144, 162, 180, 198]

```

### 3.1.3 列表元素的增加和删除

当列表增加和删除元素时，列表会自动进行内存管理，大大减少了程序员的负担。但这个特点涉及列表元素的大量移动，效率较低。除非必要，我们一般只在列表的尾部添加元素或删除元素，这会大大提高列表的操作效率。

#### append()方法

原地修改列表对象，是真正的列表尾部添加新的元素，速度最快，推荐使用。

```

>>> a=[20, 40]
>>> a.append(0)
>>> a
[20, 40, 0]
>>> a=[20, 40]
>>> a.append(80)
>>> a
[20, 40, 80]

```

#### +运算符操作

并不是真正的尾部添加元素，而是创建新的列表对象；将原列表的元素和新列表的元素依次复制到新的列表对象中。这样，会涉及大量的复制操作，对于操作大量元素不建议使用。

```

>>> a=[20, 40]
>>> id(a)
2910958151936
>>> a=a+[80]
>>> a
[20, 40, 80]
>>> id(a)
2910955783040

```

通过如上测试，我们发现变量 a 的地址发生了变化。也就是创建了新的列表对象。

#### extend()方法

将目标列表的所有元素添加到本列表的尾部，属于原地操作，不创建新的列表对象。

```
>>> a=[20, 40]
>>> id(a)
2910958085504
>>> a.extend([80, 160])
>>> a
[20, 40, 80, 160]
>>> id(a)
2910958085504
```

### insert()插入元素

使用 insert()方法可以将指定的元素插入到列表对象的任意指定位置。这样会让插入位置后面所有的元素进行移动，会影响处理速度。涉及大量元素时，尽量避免使用。类似发生这种移动的函数还有：remove()、pop()、del()，它们在删除非尾部元素时也会发生操作位置后面元素的移动。

```
>>> a=[20, 40, 160]
>>> a.insert(2, 80)
>>> a
[20, 40, 80, 160]
```

### 乘法扩展

使用乘法扩展列表，生成一个新列表，新列表元素时原列表元素的多次重复。

```
>>> a = ['sx', 100]
>>> b=a*10
>>> a
['sx', 100]
>>> b
['sx', 100, 'sx', 100, 'sx', 100, 'sx', 100, 'sx', 100, 'sx', 100, 'sx', 100, 'sx', 100]
```

### 3.1.4 列表元素的删除

#### del 删除

删除列表指定位置的元素。

```
>>> a=[100, 200, 888, 300, 400]
>>> del a[2]
>>> a
[100, 200, 300, 400]
```



### pop()方法

pop()删除并返回指定位置元素，如果未指定位置则默认操作列表最后一个元素。

```
>>> a=[10, 20, 30, 40, 50]
```

```
>>> id(a)
```

```
2910957346496
```

```
>>> a.pop()
```

```
50
```

```
>>> a
```

```
[10, 20, 30, 40]
```

```
>>> id(a)
```

```
2910957346496
```

```
>>> a.pop(1)
```

```
20
```

```
>>> a
```

```
[10, 30, 40]
```

### remove()方法

删除首次出现的指定元素，若不存在该元素抛出异常。

```
>>> a=[10, 20, 30, 40, 50, 20, 30, 20, 30]
```

```
>>> a.remove(20)
```

```
>>> a
```

```
[10, 30, 40, 50, 20, 30, 20, 30]
```

```
>>> a.remove(100)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#79>", line 1, in <module>
```

```
a.remove(100)
```

```
ValueError: list.remove(x): x not in list
```

### 3.1.5 列表元素访问和计数

#### 通过索引直接访问元素

```
>>> a=[10, 20, 30, 40, 50, 20, 30, 20, 30]
>>> a[2]
30
>>> a[10]
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
    a[10]
IndexError: list index out of range
```

#### index() 获得指定元素在列表中首次出现的索引

index() 可以获取指定元素首次出现的索引位置。语法是：index(value, [start, [end]])。其中，start 和 end 指定了搜索的范围。

```
>>> a=[10, 20, 30, 40, 50, 20, 30, 20, 30]
>>> a.index(20)
1
>>> a.index(20, 3)
5
>>> a.index(30, 5, 7)
6
```

#### count() 获得指定元素在列表中出现的次数

count() 可以返回指定元素在列表中出现的次数。

```
>>> a = [10, 20, 30, 40, 50, 20, 30, 20, 30]
>>> a.count(20)
3
```

#### len() 返回列表长度

len() 返回列表长度，即列表中元素的个数。

```
>>> a = [10, 20, 30, 40, 50, 20, 30, 20, 30]
>>> len(a)
9
```

#### 成员资格判断

判断列表中是否存在指定的元素，我们可以使用 count() 方法，返回 0 则表示不存在，返回大于 0 则表示存在。但是，一般我们会使用更加简洁的 in 关键字来判断，直接返回 True 或 False。



```
>>> a = [10, 20, 30, 40, 50, 20, 30, 20, 30]
>>> 20 in a
True
>>> 100 in a
False
>>> 20 not in a
False
```

切片操作

我们在前面学习字符串时，学习过字符串的切片操作，对于列表的切片操作和字符串类似。切片是Python序列及其重要的操作，适用于列表、元组、字符串等等。切片的格式如下：

切片slice操作可以让我们快速提取子列表或修改。标准格式为：

[起始偏移量start:终止偏移量end[:步长step]]

注：当步长省略时顺便可以省略第二个冒号

典型操作(三个量为正数的情况)如下：

操作和说明	示例	结果
[:] 提取整个列表	[10,20,30][:]	[10,20,30]
[start:]从 start 索引开始到结尾	[10,20,30][1:]	[20,30]
[:end]从头开始知道 end-1	[10,20,30][:2]	[10,20]
[start:end]从 start 到 end-1	[10,20,30,40][1:3]	[20,30]
[start:end:step]从 start 提取到 end-1，步长是 step	[10,20,30,40,50,60,70][1:6:2]	[20, 40, 60]

其他操作（三个量为负数的情况）：

示例	说明	结果
[10,20,30,40,50,60,70][-3:]	倒数三个	[50,60,70]
10,20,30,40,50,60,70][-5:- 3]	倒数第五个到倒数第三个(包头不包尾)	[30,40]
[10,20,30,40,50,60,70][::-1]	步长为负，从右到左反向提取	[70, 60, 50, 40, 30, 20, 10]

切片操作时，起始偏移量和终止偏移量不在[0,字符串长度-1]这个范围，也不会报错。起始偏移量小于0则会当做0，终止偏移量大于“长度-1”会被当成“长度-1”。例如：

```
>>> [10, 20, 30, 40][1:30]
[20, 30, 40]
```

我们发现正常输出了结果，没有报错。

复制列表所有的元素到新列表对象

如下代码实现列表元素的复制了吗？

```
>>> List1=[30, 40, 50]
>>> List2=List1
>>> id(List1)
2910955783040
>>> id(List2)
2910955783040
```

只是将List2也指向了列表对象，也就是说List1和List2持有地址值是相同的，列表对象本身的元素并没有复制。

我们可以通过如下简单方式，实现列表元素内容的复制：

```
>>> List1=[30, 40, 50]
>>> List2=[]+List1
>>> List2
[30, 40, 50]
>>> id(List1)
2910957820928
>>> id(List2)
2910958085504
```

注：我们后面也会学习copy模块，使用浅复制或深复制实现我们的复制操作。

### 3.1.6 列表排序

**修改原列表，不建新列表的排序**

```
>>> a = [20, 10, 30, 40]
>>> id(a)
2910957756224
>>> a.sort() #默认是升序
>>> a
[10, 20, 30, 40]
>>> id(a)
2910957756224
>>> a.sort(reverse=True)
>>> a
[40, 30, 20, 10]
>>> import random
>>> random.shuffle(a)
>>> a
[10, 40, 30, 20]
>>> id(a)
2910957756224
```

## 建新列表的排序

我们也可以通过内置函数sorted()进行排序，这个方法返回新列表，不对原列表做修改。

```
>>> a = [20, 10, 30, 40]
>>> id(a)
2910958087168
>>> a=sorted(a)
>>> a
[10, 20, 30, 40]
>>> id(a)
2910957346496
>>> b=sorted(a, reverse=True)
>>> b
[40, 30, 20, 10]
>>> id(b)
2910957789504
```

通过上面操作，我们可以看出，生成的列表对象a和b都是完全新的列表对象。

## reversed()返回迭代器

内置函数reversed()也支持进行逆序排列，与列表对象reverse()方法不同的是，内置函数reversed()不对原列表做任何修改，只是返回一个逆序排列的迭代器对象。

```
>>> a=[20, 10, 30, 40]
>>> c=reversed(a)
>>> c
<list_reverseiterator object at 0x000002A5C29F98E0>
>>> list(c)
[40, 30, 10, 20]
>>> list(c)
[]
```

我们打印输出c发现提示是：list\_reverseiterator。也就是一个迭代对象。同时，我们使用list(c)进行输出，发现只能使用一次。第一次输出了元素，第二次为空。那是因为迭代对象在第一次时已经遍历结束了，第二次不能再使用。

## 3.1.7列表相关的其他内置函数汇总

### max和min

用于返回列表中最大和最小值。

```
>>> a = [10, 3, 5, 100, 66]
>>> max(a)
100
>>> min(a)
3
```

## Sum

对数值型列表的所有元素进行求和操作，对非数值型列表运算则会报错。

```
>>> a = [10, 3, 5, 100, 66]
>>> sum(a)
184
```

## 3.1.8 多维列表

### 二维列表

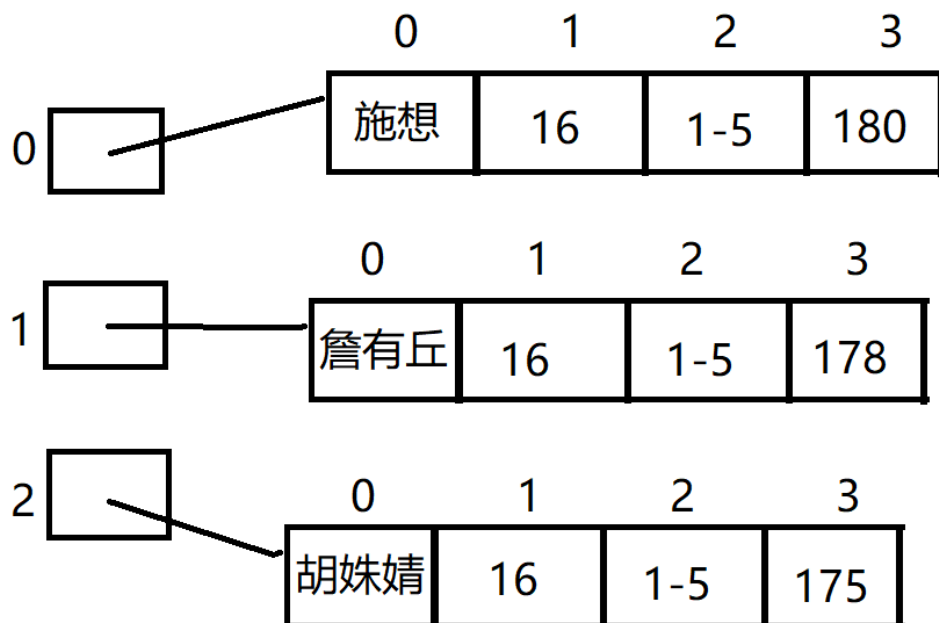
一维列表可以帮助我们存储一维、线性的数据。

二维表格可以帮助我们存储二维、表格的数据。

姓名	年龄	班级	身高
施想	16	1-5	180
詹有丘	16	1-5	178
胡姝婧	17	2-5	175

```
>>> a = [
    ["施想", 16, "1-5", 180],
    ["詹有丘", 16, "1-5", 178],
    ["胡姝婧", 17, "2-5", 175],
]
>>> a
[['施想', 16, '1-5', 180], ['詹有丘', 16, '1-5', 178], ['胡姝婧', 17, '2-5', 175]]
```

内存结构图：



```
>>> print(a[1][0], a[1][1], a[1][2])
詹有丘 16 1-5
```

## 3.2 元组

列表属于可变序列，可以任意修改列表中的元素。元组属于不可变序列，不能修改元组中的元素。因此，元组没有增加元素、修改元素、删除元素相关的方法。

因此，我们只需要学习元组的创建和删除，元组中元素的访问和计数即可。元组支持如下操作：

1. 索引访问
2. 切片操作
3. 连接操作
4. 成员关系操作
5. 比较运算操作
6. 计数：元组长度len()、最大值max()、最小值min()、求和sum()等

### 3.2.1 元组的创建

1. 通过()创建元组。小括号可以省略。

`a = (10,20,30)` 或者 `a = 10,20,30`

如果元组只有一个元素，则必须后面加逗号。这是因为解释器会把(1)解释为整数1，(1,)解释为元组。

```
>>> a=(1)
>>> type(a)
<class 'int'>
>>> a=(1,)
>>> type(a)
<class 'tuple'>
```

2. 通过tuple()创建元组

tuple(可迭代的对象)

```
>>> b=tuple()
>>> b=tuple("abc")
>>> b=tuple(range(3))
>>> b=tuple([2, 3, 4])
```

总结：

tuple()可以接收列表、字符串、其他序列类型、迭代器等生成元组。

list()可以接收元组、字符串、其他序列类型、迭代器等生成列表。

### 3.2.2 元组的元素访问和计数

1. 元组的元素不能修改
2. 元组的元素访问和列表一样，只不过返回的仍然是元组对象。

```
>>> a=(20, 10, 30, 9, 8)
>>> a[1]
10
>>> a[1:3]
(10, 30)
>>> a[:4]
(20, 10, 30, 9)
```

3. 列表关于排序的方法`list.sorted()`是修改原列表对象，元组没有该方法。如果要对元组排序，只能使用内置函数`sorted(tupleObj)`，并生成新的列表对象。

```
>>> a = (20, 10, 30, 9, 8)
>>> sorted(a)
[8, 9, 10, 20, 30]
```

### 3.2.3 zip

`zip(列表1, 列表2, ...)`将多个列表对应位置的元素组合成为元组，并返回这个zip对象。

```
>>> a=[10, 20, 30]
>>> b=[40, 50, 60]
>>> c=[70, 80, 90]
>>> d=zip(a, b, c)
>>> list(d)
[(10, 40, 70), (20, 50, 80), (30, 60, 90)]
```

### 元组总结

1. 元组的核心特点是：不可变序列。
2. 元组的访问和处理速度比列表快。
3. 与整数和字符串一样，元组可以作为字典的键，列表则永远不能作为字典的键使用。

## 3.3字典

字典是“键值对”的无序可变序列，字典中的每个元素都是一个“键值对”，包含：“键对象”和“值对象”。可以通过“键对象”实现快速获取、删除、更新对应的“值对象”。

列表中我们通过“下标数字”找到对应的对象。字典中通过“键对象”找到对应的“值对象”。“键”是任意的不可变数据，比如：整数、浮点数、字符串、元组。但是：列表、字典、集合这些可变对象，不能作为“键”。并且“键”不可重复。

“值”可以是任意的数据，并且可重复。

一个典型的字典的定义方式：

```
a = {'name':'shixiang','age':16,'job':'student'}
```

### 3.3.1字典的创建

1. 我们可以通过{}、dict()来创建字典对象。

```
>>> a = {'name': 'sx', 'age': 16, 'job': 'student'}
>>> b = dict(name='sx', age=16, job='student')
>>> a = dict([("name", "sx"), ("age", 16)])
>>> c = {}
>>> d = dict()
```

2. 通过zip()创建字典对象

```
>>> k = ['name', 'age', 'job']
>>> v = ['sx', 16, 'student']
>>> d = dict(zip(k, v))
>>> d
{'name': 'sx', 'age': 16, 'job': 'student'}
```

3. 通过fromkeys创建值为空的字典

```
>>> a = dict.fromkeys(['name', 'age', 'job'])
>>> a
{'name': None, 'age': None, 'job': None}
```

### 3.3.2字典元素的访问

为了测试各种访问方法，我们这里设定一个字典对象：

```
a = {'name': 'sx', 'age': 16, 'job': 'student'}
```

1. 通过[键]获得“值”。若键不存在，则抛出异常。

```
>>> a = {'name': 'sx', 'age': 16, 'job': 'student'}
>>> a['name']
'sx'
>>> a['age']
16
>>> a['sex']
```

```
Traceback (most recent call last):
  File "<pyshell#193>", line 1, in <module>
    a['sex']
KeyError: 'sex'
```

2. 通过get()方法获得“值”。推荐使用。优点是：指定键不存在，返回None；也可以设定指定键不存在时默认返回的对象。推荐使用get()获取“值对象”。

```
>>> a.get('name')
'sx'
>>> a.get('sex')
None
>>> a.get('sex', 'male')
'male'
```



## 3. 列出所有的键值对

```
>>> a.items()
dict_items([('name', 'sx'), ('age', 16), ('job', 'student')])
```

## 4. 列出所有的键，列出所有的值

```
>>> a.keys()
dict_keys(['name', 'age', 'job'])
>>> a.values()
dict_values(['sx', 16, 'student'])
```

## 5. len() 键值的个数

## 6. 检测一个“键”是否在字典中

```
>>> a = {"name": "sx", "age": 16}
>>> 'name' in a
True
```

## 3.3.3字典元素添加、修改、删除

1. 给字典新增“键值对”。如果“键”已经存在，则覆盖旧的键值对；如果“键”不存在，则新增“键值对”。

```
>>> a = {'name': 'sx', 'age': 16, 'job': 'student'}
>>> a['address'] = 'JDFZ'
>>> a['age'] = 16
>>> a
{'name': 'sx', 'age': 16, 'job': 'student', 'address': 'JDFZ'}
```

2. 使用update()将新字典中所有键值对全部添加到旧字典对象上。如果key有重复，则直接覆盖。

```
>>> a = {'name': 'sx', 'age': 16, 'job': 'student'}
>>> b = {'name': '施想', 'money': 1000, 'sex': 'male'}
>>> a.update(b)
>>> a
{'name': '施想', 'age': 16, 'job': 'student', 'money': 1000, 'sex': 'male'}
```

3. 字典中元素的删除，可以使用del()方法；或者clear()删除所有键值对；pop()删除指定键值对，并返回对应的“值对象”；

```
>>> a = {'name': 'sx', 'age': 16, 'job': 'student'}
>>> del(a['name'])
>>> a
{'age': 16, 'job': 'student'}
>>> b = a.pop('age')
>>> b
16
```



### 3.4 序列解包

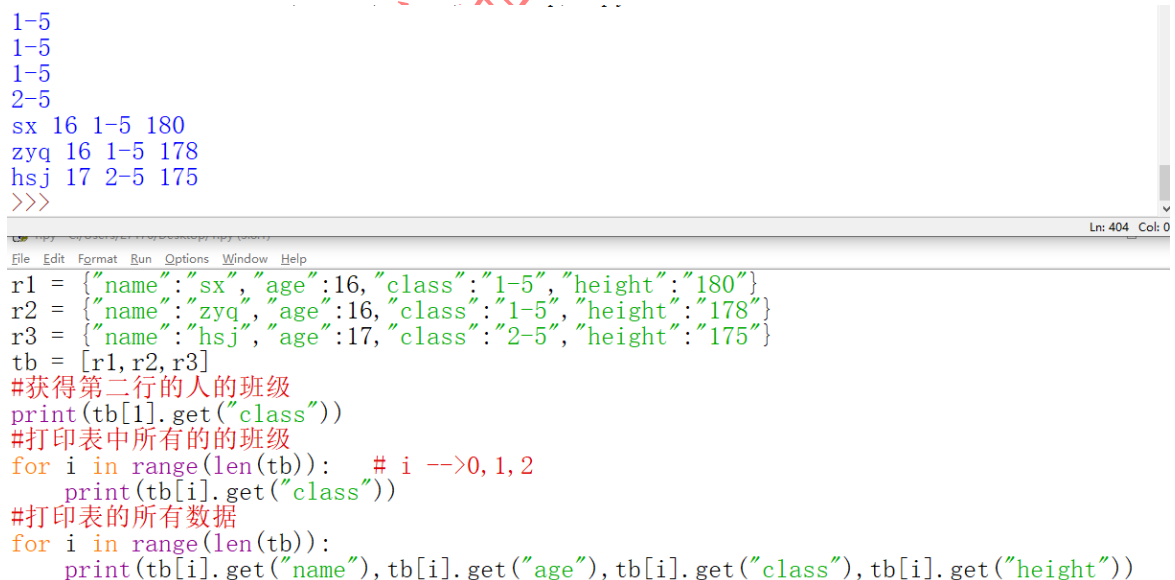
序列解包可以用于元组、列表、字典。序列解包可以让我们方便的对多个变量赋值。

```
>>> x, y, z = (20, 30, 10)
>>> x
20
>>> y
30
>>> z
10
>>> (a, b, c) = (9, 8, 10)
>>> a
9
>>> [a, b, c] = [10, 20, 30]
>>> a
10
>>> b
20
```

序列解包用于字典时，默认是对“键”进行操作；如果需要对键值对操作，则需要使用items()；如果需要对“值”进行操作，则需要使用values()；

### 3.5 表格数据使用字典和列表存储，并实现访问

```
1-5
1-5
1-5
2-5
sx 16 1-5 180
zyq 16 1-5 178
hsj 17 2-5 175
>>>
```



```
r1 = {"name": "sx", "age": 16, "class": "1-5", "height": "180"}
r2 = {"name": "zyq", "age": 16, "class": "1-5", "height": "178"}
r3 = {"name": "hsj", "age": 17, "class": "2-5", "height": "175"}
tb = [r1, r2, r3]
#获得第二行的人的班级
print(tb[1].get("class"))
#打印表中所有的班级
for i in range(len(tb)): # i --> 0, 1, 2
    print(tb[i].get("class"))
#打印表的所有数据
for i in range(len(tb)):
    print(tb[i].get("name"), tb[i].get("age"), tb[i].get("class"), tb[i].get("height"))
```

### 3.6 字典核心底层原理(重要)

字典对象的核心是散列表。散列表是一个稀疏数组（总是有空白元素的数组），数组的每个单元叫做bucket。每个bucket有两部分：一个是键对象的引用，一个是值对象的引用。由于，所有bucket结构和大小一致，我们可以通过偏移量来读取指定bucket。

0	key1	value1
1		
2	key2	value2
3		
4		
5	key4	value4

bucket

## 将一个键值对放进字典的底层过程

```
>>> a = {}
```

```
>>> a["name"] = "sx"
```

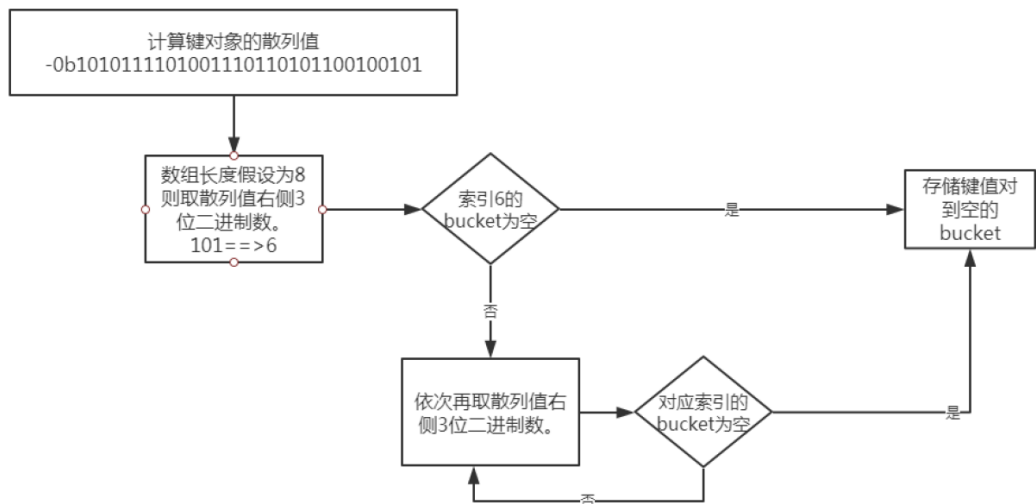
假设字典a对象创建完后，数组长度为8：

0	
1	
2	
3	
4	
5	
6	
7	

我们要把"key"="sx"这个键值对放到字典对象a中，首先第一步需要计算键"key"的散列值。Python中可以通过hash()来计算。>>> bin(hash("key"))

```
>>> bin(hash("key"))
'-0b10101111010011101101101100100101'
```

由于数组长度为8，我们可以拿计算出的散列值的最右边3位数字作为偏移量，即“101”，十进制是数字5。我们查看偏移量5，对应的bucket是否为空。如果为空，则将键值对放进去。如果不为空，则依次取右边3位作为偏移量，即“100”，十进制是数字4。再查看偏移量为4的bucket是否为空。直到找到为空的bucket将键值对放进去。流程图如下：



## 根据键查找“键值对”的底层过程

我们明白了，一个键值对是如何存储到数组中的，根据键对象取到值对象，理解起来就简单了。

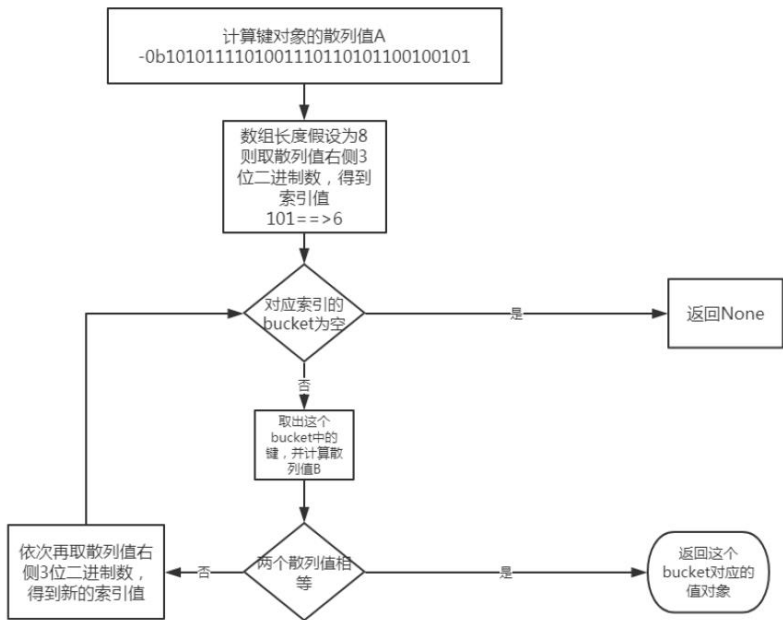
```
>>> a.get("name")
'sx'
```

当我们调用`a.get("name")`，就是根据键“name”查找到“键值对”，从而找到值对象“sx”。

第一步，我们仍然要计算“name”对象的散列值：

```
>>> bin(hash("name"))
'-0b10101111101001110110101100100101'
```

和存储的底层流程算法一致，也是依次取散列值的不同位置的数字。假设数组长度为8，我们可以拿计算出的散列值的最右边3位数字作为偏移量，即“101”，十进制是数字5。我们查看偏移量5，对应的bucket是否为空。如果为空，则返回None。如果不为空，则将这个bucket的键对象计算对应散列值，和我们的散列值进行比较，如果相等。则将对应“值对象”返回。如果不相等，则再依次取其他几位数字，重新计算偏移量。依次取完后，仍然没有找到。则返回None。流程图如下：



用法总结：

1. 键必须可散列

(1) 数字、字符串、元组，都是可散列的。

(2) 自定义对象需要支持下面三点：

① 支持hash()函数

② 支持通过`__eq__()`方法检测相等性。

③ 若`a==b`为真，则`hash(a)==hash(b)`也为真。

2. 字典在内存中开销巨大，典型的空间换时间。

3. 键查询速度很快

4. 往字典里面添加新建可能导致扩容，导致散列表中键的次序变化。因此，不要在遍历字典的同时进行字典的修改。

### 3.7 集合

集合是无序可变，元素不能重复。实际上，集合底层是字典实现，集合的所有元素都是字典中的“键对象”，因此是不能重复的且唯一的。

#### 集合创建和删除

1. 使用`{}`创建集合对象，并使用`add()`方法添加元素

```
{3, 5, 7}
>>> a.add(9)
>>> a
{9, 3, 5, 7}
```

2. 使用`set()`，将列表、元组等可迭代对象转成集合。如果原来数据存在重复数据，则只保留一个。

```
>>> a=[1, 2, 3, 4, 2]
>>> b=set(a)
>>> b
{1, 2, 3, 4}
```

# 控制语句

我们在前面学习的过程中，都是很短的示例代码，没有进行复杂的操作。现在，我们将开始学习流程控制语句。

前面学习的变量、数据类型（整数、浮点数、布尔）、序列（字符串、列表、元组、字典、集合），可以看做是数据的组织方式。数据可以看做是“砖块”！

流程控制语句是代码的组织方式，可以看做是“混凝土”。

一个完整的程序，离不开“砖块”，也离不开“混凝土”。他们的组合，才能让我们建立从小到“一个方法”，大到“操作系统”，这样各种各样的“软件”。

## 4.1 选择结构

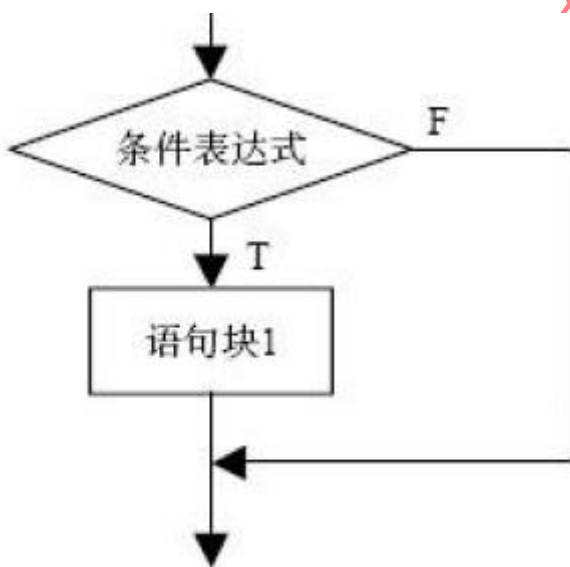
完全按照设计好的流程执行程序，几乎是太完美了。然后现实生活更多的时候需要流程选择，因此流程控制至关重要。日常生活中大家都会有意识无意识的用到流程控制，用来判断自己下一步的行动计划。例如，今日行程是去新华书店买书还是去游泳馆游泳呢，需要作出选择。

Python语言的流程控制包括三大类：

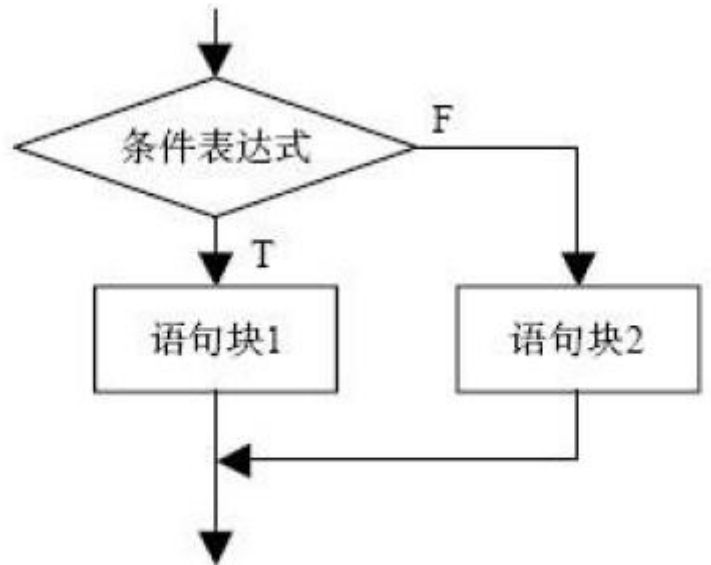
1. 判断
2. 循环
3. 中断/继续

选择结构通过判断条件是否成立，来决定执行哪个分支。选择结构有多种形式，分为：单分支、双分支、多分支。流程图如下：

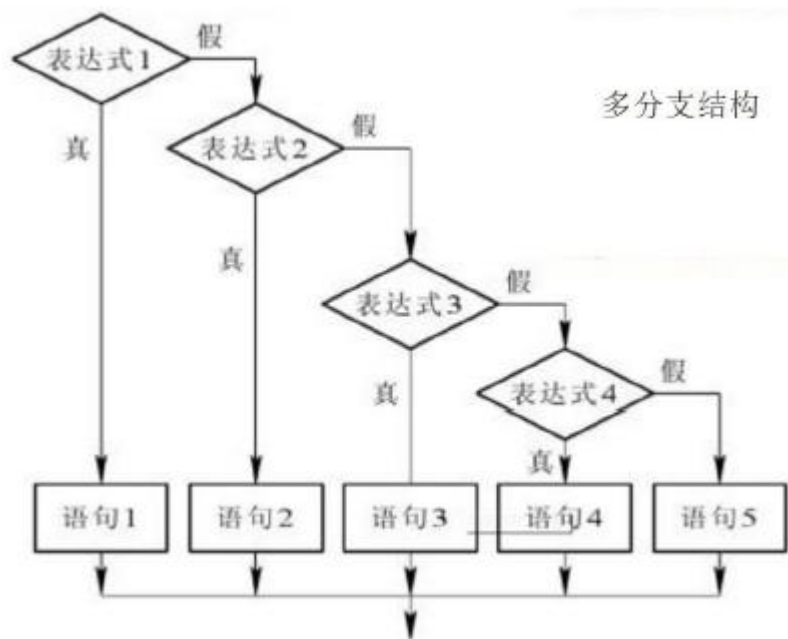
单分支结构



双分支结构



以下是多分支结构：



### 单分支选择结构

if语句单分支结构的语法形式如下：

if 条件表达式:

    语句/语句块

其中：

② . 条件表达式：可以是逻辑表达式、关系表达式、算术表达式等等。

②. 语句/语句块：可以是一条语句，也可以是多条语句。多条语句，缩进必须对齐一致。

### 条件表达式详解

在选择和循环结构中，条件表达式的值为False的情况如下：

False、0、0.0、空值、None、空序列对象（空列表、空元祖、空集合、空字典、空字符串）、空range对象、空迭代对象。

其他情况，均为True。这么看来，Python所有的合法表达式都可以看做条件表达式，甚至包括函数调用的表达式。

### 条件表达式中，不能有赋值操作符“=”

在Python中，条件表达式不能出现赋值操作符“=”，避免了其他语言中经常误将关系运算符“==”写作赋值运算符“=”带来的困扰。代码将会报语法错误。

### 双分支选择结构

双分支结构的语法格式如下：

if 条件表达式:

    语句1/语句块1

else:

    语句2/语句块2

### 三元条件运算符

Python提供了三元运算符，用来在某些简单双分支赋值情况。三元条件运算符语法格式如下：

条件为真时的值if (条件表达式) else条件为假时的值

多分支选择结构：

多分支选择结构的语法格式如下：

if 条件表达式1：

语句1/语句块1

elif 条件表达式2：

语句2/语句块2

elif 条件表达式n：

语句n/语句块n

[else:

语句n+1/语句块n+1

]

### 选择结构嵌套

选择结构可以嵌套，使用时一定要注意控制好不同级别代码块的缩进量，因为缩进量决定了代码的从属关系。语法格式如下：

if 表达式 1：

语句块 1

if 表达式 2：

语句块 2

else:

语句块 3

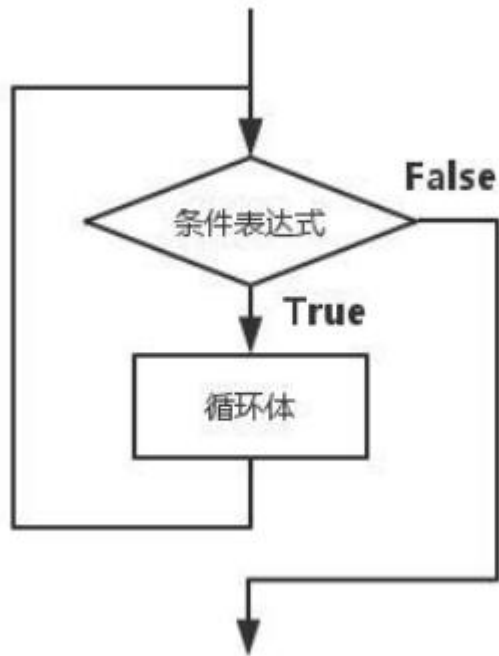
else:

if 表达式 4：

语句块 4

## 4.2 循环结构

循环结构用来重复执行一条或多条语句。表达这样的逻辑：如果符合条件，则反复执行循环体里的语句。在每次执行完后都会判断一次条件是否为True，如果为True则重复执行循环体里的语句。图示如下：



循环体里面的语句至少应该包含改变条件表达式的语句,以使循环趋于结束;否则,就会变成一个死循环。

### while循环

while循环的语法格式如下:

```
while 条件表达式:
    循环体语句
```

### for循环和可迭代对象遍历

for循环通常用于可迭代对象的遍历。for循环的语法格式如下:

```
for 变量 in 可迭代对象:
    循环体语句
```

### 可迭代对象

Python包含以下几种可迭代对象:

1. 序列。包含: 字符串、列表、元组
2. 字典
3. 迭代器对象 (iterator)
4. 生成器函数 (generator)
5. 文件对象

我们已经在前面学习了序列、字典等知识,迭代器对象和生成器函数将在后面进行详解。

### range对象

range对象是一个迭代器对象,用来产生指定范围的数字序列。格式为:

```
range(start, end [,step])
```

生成的数值序列从start开始到end结束(不包含end)。若没有填写start,则默认从0开始。

step是可选的步长,默认为1。如下是几种典型示例:



for i in range(10)产生序列：0 1 2 3 4 5 6 7 8 9

for i in range(3,10)产生序列：3 4 5 6 7 8 9

for i in range(3,10,2)产生序列：3 5 7 9

嵌套循环和综合练习

一个循环体内可以嵌入另，般称为“套”，或者多重”。

## break语句

break语句可用于while和for循环，用来结束整个循环。当有嵌套循环时，break语句只能跳出最近一层的循环。

## continue语句

continue语句用于结束本次循环，继续下一次。多个循环嵌套时，continue也是应用于最近的一层循环。

## else语句

while、for循环可以附带一个else语句（可选）。如果for、while语句没有被break语句结束，则会执行else子句，否则不执行。语法格式如下：

while 条件表达式：

循环体

else:

语句块

或者：

for 变量 in 可迭代对象：

循环体

else:

语句块

## 循环代码优化

虽然计算机越来越快，空间也越来越大，我们仍然要在性能问题上“斤斤计较”。编写循环时，遵守下面三个原则可以大大提高运行效率，避免不必要的低效计算：

1. 尽量减少循环内部不必要的计算
2. 嵌套循环中，尽量减少内层循环的计算，尽可能向外提。
3. 局部变量查询较快，尽量使用局部变量