

第2章

基础语法

本章内容

01 变量的定义

02 Number数据类型

03 String数据类型

04 List数据类型

05 Tuple数据类型

06 Set数据类型

07 Dictionary数据类型

08 占位运算符和算术运算符

09 赋值运算符、比较运算符
和逻辑运算符

10 位运算符

11 身份运算符和成员运算符

12 序列运算符和运算符优先级

13 条件语句概述

14 条件语句实现和pass

15 循环语句概述和for循环

16 while循环和索引

17 break、continue和else

变量的定义

表示数据的量的分类



常量

是指在程序运行过程中值不能发生改变的量

如1、3.5、 $3+4i$ 、"abc"等

表示数据的量的分类



变量

是指在程序运行过程中值可以发生改变的量。

与数学中的变量一样，需要为Python中的每一个变量指定一个名字，如x、y、test等。

Python是一种弱类型的语言，变量的类型由其值的类型决定。

Python变量在使用前不需要先定义，为一个变量赋值后，则该变量会自动创建。

变量的命名规则

变量名可以包括字母、数字和下划线，但是数字不能作为开头字符

◆ 例如，test1是有效变量名，而1test则是无效变量名

系统关键字不能做变量名使用

◆ 例如，and、break等都是系统关键字，不能作为变量名使用

Python的变量名区分大小写

◆ 例如，test和Test是两个不同的变量

提示 Python 3.x默认使用UTF-8编码，变量名中允许包含中文，如“测试”是一个有效的变量名。

变量定义和使用示例



变量定义和使用示例

4



`print(Test)` #输出 "123"

5



`test=10.5` #浮点型变量

6



`print(test)` #输出 "10.5 "

同时定义多个变量



语法格式

变量1,变量2,...,变量

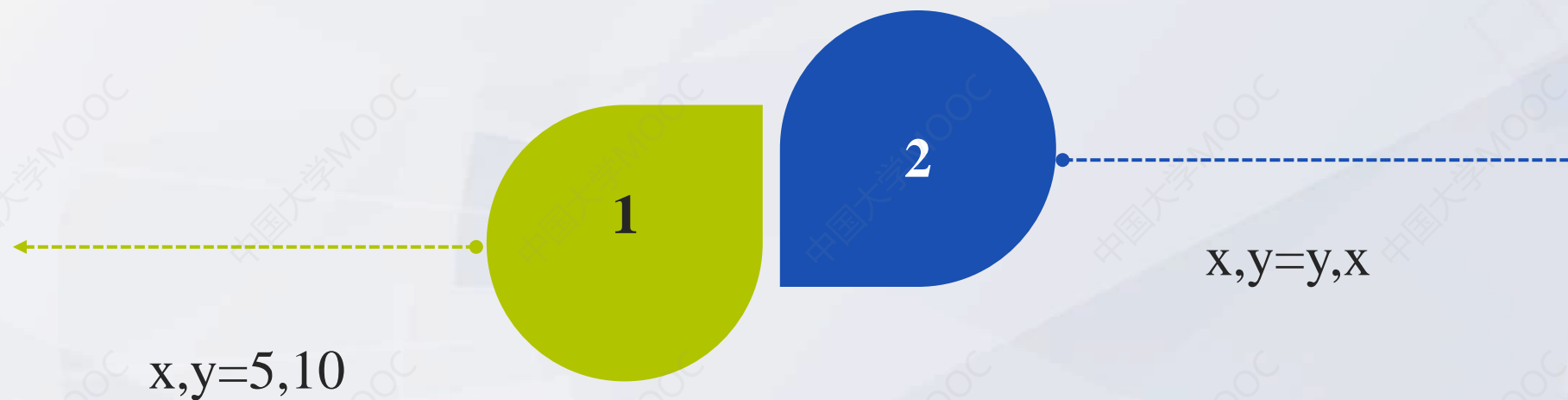
n=值1,值2,...,值n



例

name,age='张三',18

修改多个变量的值



修改多个变量的值



提示

对于赋值运算，会先计算赋值运算符右边的表达式的值，再将计算结果赋给左边的变量。

因此，第2行代码会先将赋值运算符右侧的y和x的值得到，再将它们分别赋给左边的变量。

取出右侧的y和x的值后，第2行代码转换为 “x,y=10,5” ，然后再执行赋值运算，即将10赋给x、将5赋给y。

Number数据类型

数据类型概述

一种编程语言所支持的数据类型决定了该编程语言所能保存的数据

Python语言常用的内置数据类型

◆ Number（数字）、String（字符串）、List（列表）、Tuple（元组）、Set（集合）、Dictionary（字典）

Python中有3种不同的数字类型

◆ int（整型）、float（浮点型）、complex（复数类型）。

整型

整型数字包括正整数、0和负整数，不带小数点，无大小限制



整数可以使用不同的进制来表示

不加任何前缀为十进制整数

加前缀0o为八进制整数

加前缀0x则为十六进制整数


整型



例

`a,b,c=10,0o10,0x10` #a、 b、 c的值分别为10、 8、 16

提示

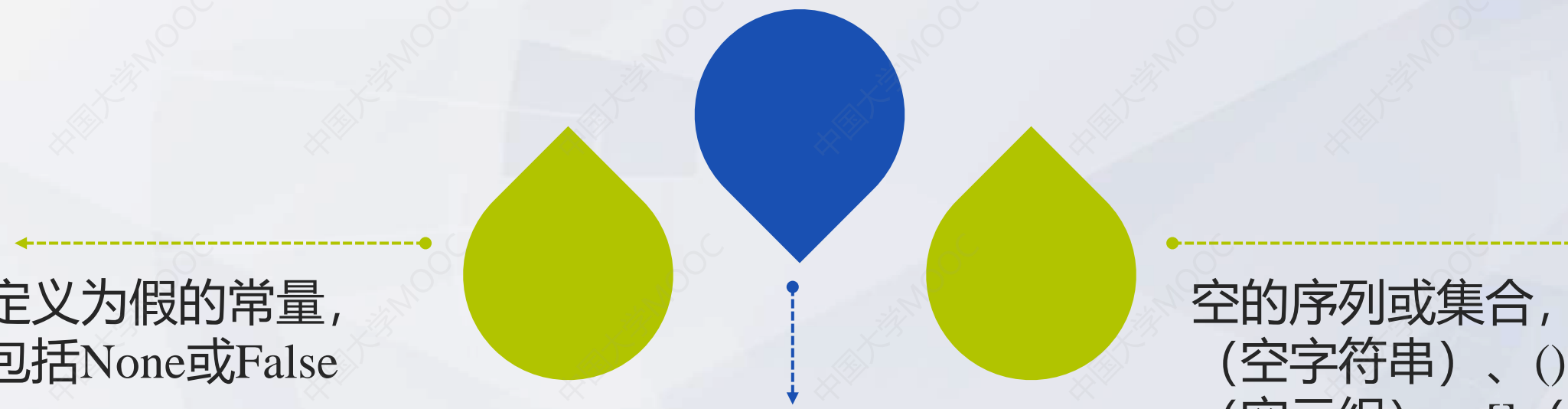


Python语言中提供了 Boolean（布尔）类型，用于表示逻辑值 True（逻辑真）和 False（逻辑假）

Boolean 类型是整型的子类型，在作为数字参与运算时，False 自动转为 0，True 自动转为 1

提示

使用bool函数可以将其他类型的数据转为Boolean类型，当给bool函数传入下列参数时其将会返回False：



定义为假的常量，
包括None或False

任意值为0的数值，
如0、0.0、0j等

空的序列或集合，如"
(空字符串)、() (空元组)、[] (空列表) 等

浮点型

浮点型数字使用C语言中的double类型实现，可以用来表示实数

- ◆ 如3.14159、-10.5、3.25e3等
- ◆ 3.25e3是科学记数法的表示方式，其中e表示10，因此，3.25e3实际上表示的浮点数是 $3.25 \times 10^3 = 3250.0$

查看浮点数的取值范围和精度的代码示例

```
import sys #导入sys包
```

```
sys.float_info #查看当前环境中浮点型数字的取值范围和精度
```

浮点型

min和max是浮点数的最小值和最大值，dig是浮点数所能精确表示的十进制数字的最大位数

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,  
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,  
min_10_exp=-307, dig=15, mant_dig=53,  
epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

复数类型



复数由实部和虚部组成，每一部分都是一个浮点数，其书写方法如下：

$a+bj$ 或 $a+bJ$

其中， a 和 b 是两个数字， j 或 J 是虚部的后缀，即 a 是实部、 b 是虚部



在生成复数时，也可以使用`complex`函数，其语法格式如下：

`complex([real[,imag]])`

其中，`real`为实部值，`imag`为虚部值，返回值为`real+imag*1j`

复数类型



提示

如果省略虚部imag的值,
则返回的复数为 $\text{real}+0j$;
如果实部real和虚部imag
的值都省略, 则返回的复
数为 $0j$



例如，对于下面的代码：

```
c1,c2,c3,c4,c5=3+5.5j,3.25e3j,complex(5,-3.5),  
complex(5),complex()
```

执行完毕后，c1、c2、c3、c4和c5的值分别是：
(3+5.5j)、3250j、(5-3.5j)、(5+0j)和0j

String数据类型

概述

Python语言中只有用于保存字符串的String类型，而没有用于保存单个字符的数据类型

Python中的字符串可以写在一对单引号中，也可以写在一对双引号或一对三双引号中

三种写法的区别将在后面介绍，目前我们使用一对单引号或一对双引号的写法

对于不包含任何字符的字符串，如"（一对单引号）或""（一对双引号），称为空字符串（或简称为空串）

概述



例如：

```
s1,s2='Hello World!',"你好，世界！"
```

执行完毕后，s1和s2的值分别是字符串 "Hello World!" 和 "你好，世界！"

字符串转成整数

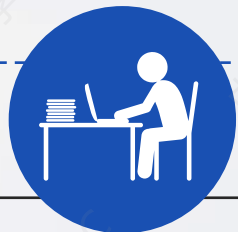


int函数

第一个参数是要转换为整数的字符串（要求必须是一个整数字符串，否则会报错）

第二个参数是字符串中整数的数制（不指定则默认为10）

字符串转成整数



例如

`int('35')`返回整数35,
`int('35',8)`返回整数29,
`int('35+1')`则会因无法转换
而报错



提示

`int`函数仅是将字符串中的
数字直接转为整数，不会
做任何运算。当要转换的
字符串是一个包含运算的
表达式时，`int`函数会报错

字符串转成浮点数

例如

float函数

提示

只有一个参数，即要转换为浮点数的字符串（要求必须是一个整数或浮点数字符串）

float('35') 返回浮点数 35.0 ,
float('35.5') 返回浮点数 35.5 ,
float('35.5+3') 则会因无法转换而报错

与int函数类似，float函数仅是将字符串中的数字直接转为浮点数，不会做任何运算。当要转换的字符串是一个包含运算的表达式时，float函数会报错

常用转义字符

转义字符	描述	转义字符	描述
\ (在行尾时)	续行符	\n	换行
\\	反斜杠符号	\r	回车
\'	单引号	\t	制表符
\"	双引号		

常用转义字符

例如：

1

▶ `s1='Hello \`

2

▶ `World!'` #上一行以\作为行尾，说明上一行与当前行是同一条语句

3

▶ `s2='It's a book.'` #单引号非成对出现，报 `SyntaxError` 错误

常用转义字符



常用转义字符

执行完毕后，使用print函数依次输出成功创建的各变量的值，则可以得到如下结果：

s1输出 “Hello World” ；

s2没有创建成功，所以会报NameError错误；

s3和s4都输出 “It's a book.” ；

s5输出两行信息，第一行输出 “你好！” ，第二行输出 “欢迎学习Python语言程序设计！”

子串截取

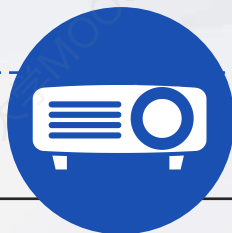


利用下标 “[]” 可以从字符串中截取一个子串，其语法格式为：

`s[beg:end]`

其中，`s`为原始字符串，`beg`是要截取子串在`s`中的起始下标，`end`是要截取子串在`s`中的结束下标。

子串截取



提示：

截取子串中包含的字符是s中从beg至end-1（不包括end）位置上的字符；

省略beg，则表示从s的开始字符进行子串截取，等价于s[0:end]；

省略end，则表示截取的子串中包含从beg位置开始到最后一个字符之间的字符（包括最后一个字符）；

beg和end都省略则表示子串中包含s中的所有字符。

下标索引方式

从前向后
索引

第1个字符的下标为0，其他字符的下标是前一个字符的下标增1

从后向前
索引

从后向前索引方式中，最后一个字符的下标为-1，其他字符的下标是后一个字符的下标减1

在截取子串时，既可以只使用某一种下标索引方式，也可以同时使用两种下标索引方式

下标索引方式

字符串	欢	迎	学	习	P	y	t	h	o	n	语	言	程	序	设	计	！
从前向后 索引	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
从后向前 索引	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

下标索引方式

例如：



1

```
s='欢迎学习Python语言程序设计！'
```



2

```
print(s[2:4]) #输出 “学习”
```



3

```
print(s[-3:-1]) #输出 “设计”
```

下标索引方式

4

▶ `print(s[2:-1])` #输出 “学习Python语言程序设计”

5

▶ `print(s[:10])` #输出 “欢迎学习Python”

6

▶ `print(s[-5:])` #输出 “程序设计！ ”

7

▶ `print(s[:])` #输出 “欢迎学习Python语言程序设计！ ”

截取单一字符



也可以采用下面的写法：

```
s[idx]
```

其中，idx是要截取的字符的下标。



例如：

```
1 s='欢迎学习Python语言程序设计！'
```

```
2 print(s[2]) #输出 “学”
```

```
3 print(s[-1]) #输出 “！”
```

截取单一字符

注意

使用下标 “[]” 可以访问字符串中的元素，但不能修改。例如，对于 “s[2]='复’” 这样的代码，执行时会报TypeError错误。

list数据类型

概述

List（列表）是Python中一种非常重要的数据类型。

列表中可以包含多个元素，且元素类型可以不相同。

每一元素可以是任一数据类型，包括列表（即列表嵌套）及后面要介绍的元组、集合、字典。

所有元素都写在一对方括号“[]”中，每两个元素之间用逗号分隔。

对于不包含任何元素的列表，即[]，称为空列表。

列表元素索引

列表中元素的索引方式与字符串中元素的索引方式完全相同，也支持从前向后索引和从后向前索引两种方式



与字符串相同，利用下标 “[]” 可以从已有列表中取出其中部分元素形成一个新列表，其语法格式为：

```
ls[beg:end]
```

其中，beg是要取出的部分元素在ls中的起始下标，end是要取出的部分元素在ls中的结束下标。

列表元素索引



提示

省略beg，则表示从ls中的第一个元素开始，等价于ls[0:end]；省略end，则表示要取出的部分元素从beg位置开始一直到最后一个元素（包括最后一个元素）；beg和end都省略则取出ls中的所有元素。

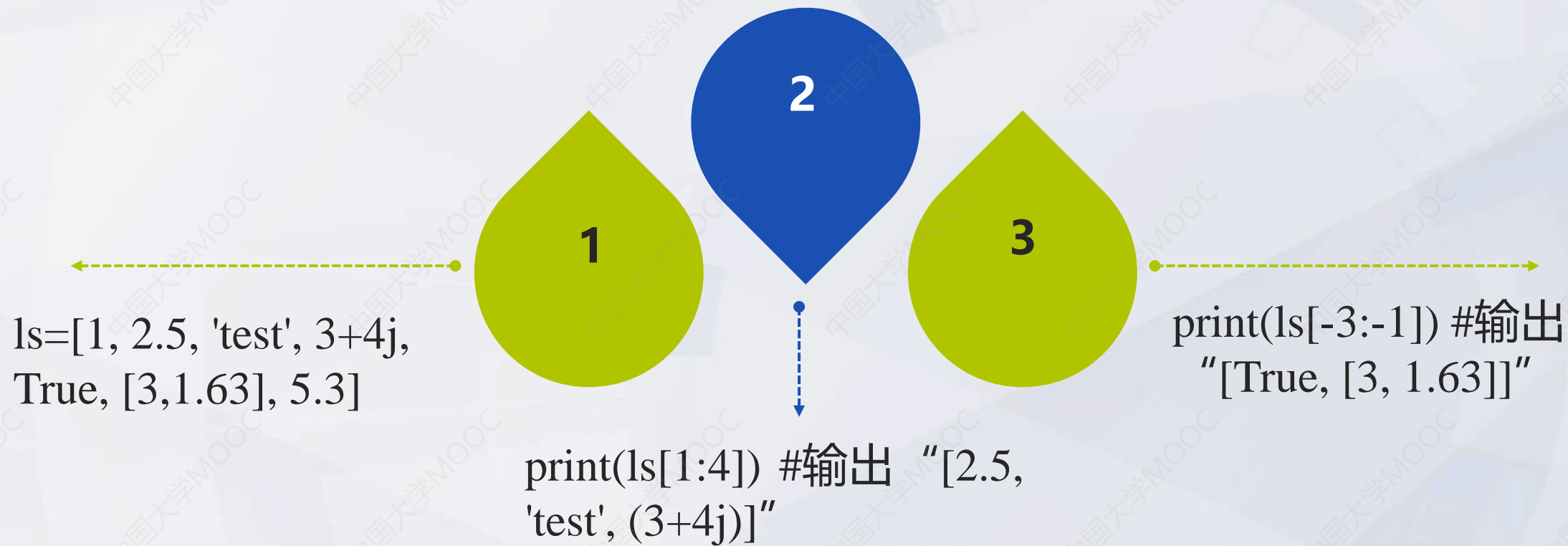
列表元素索引

例如

对于ls=[1, 2.5, 'test', 3+4j, True, [3,1.63], 5.3]这个列表，其各元素的下标为：

列表	1	2.5	'test'	3+4j	True	[3,1.63]	5.3
从前向后索引	0	1	2	3	4	5	6
从后向前索引	-7	-6	-5	-4	-3	-2	-1

列表元素索引



列表元素索引

4

▶ `print(ls[2:-1])` #输出 `"['test', (3+4j), True, [3, 1.63]]"`

5

▶ `print(ls[:3])` #输出 `"[1, 2.5, 'test']"`

6

▶ `print(ls[-2:])` #输出 `"[[3, 1.63], 5.3]"`

7

▶ `print(ls[:])` #输出 `"[1, 2.5, 'test', (3+4j), True, [3, 1.63], 5.3]"`

访问单个元素



如果只访问列表`ls`中的某一个元素，则可以使用下面的写法：

```
ls[idx]
```

其中，`idx`是要访问的元素的下标。



例如：

```
1 ls=[1, 2.5, 'test', 3+4j, True, [3,1.63], 5.3]
```

```
2 print(ls[2]) #输出 "test"
```

```
3 print(ls[-3]) #输出 "True"
```

访问单个元素

例如

注意

`ls[beg:end]` 返回的仍然是一个列表；而 `ls[idx]` 返回的是列表中的一个元素。

可见

`ls[2:3]` 返回的是只有一个字符串元素 `'test'` 的列表，而 `ls[2]` 返回的则是 `ls` 中第3个元素的值（即字符串 `'test'`）。

对于 `ls=[1, 2.5, 'test', 3+4j, True, [3,1.63], 5.3]`，通过 `"print(ls[2:3])"` 和 `"print(ls[2])"` 输出的结果分别是 `"['test']"` 和 `"test"`。

元素修改

通过下标 “[]” 不仅可以访问列表中的某个元素，还可以对元素进行修改。例如：

1

▶ `ls=[1, 2.5, 'test', 3+4j, True, [3,1.63], 5.3]`

2

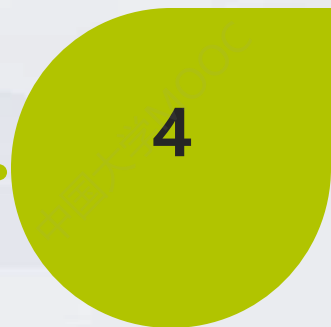
▶ `print(ls)` #输出 “[1, 2.5, 'test', (3+4j), True, [3, 1.63], 5.3]”

3

▶ `ls[2]=15` #将列表ls中第3个元素的值改为15

元素修改

`print(ls)` #输出 “[1, 2.5, 15, (3+4j), True, [3, 1.63], 5.3]”



`ls[1:4]=['python',20]` #将列表ls中第2至4个元素替换为['python',20]中的元素

元素修改



6

```
print(ls) #输出 "[1, 'python', 20, True, [3, 1.63], 5.3]"
```



7

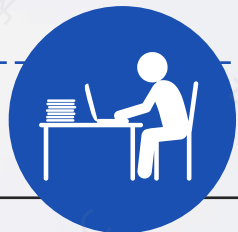
```
ls[2]=['program',23.15] #将列表ls中第3个元素替换为['program',23.15]
```



8

```
print(ls) #输出 "[1, 'python', ['program', 23.15], True, [3, 1.63], 5.3]"
```

元素修改



9

`ls[0:2]=[]` #将列表ls中前两个元素替换为空列表[], 即将前两个元素删除



10

`print(ls)` #输出 `"['program', 23.15], True, [3, 1.63], 5.3]"`

元素修改



注意：

在对列表中的元素赋值时，既可以通过 `ls[idx]=a` 这种方式修改单个元素的值，也可以通过 `ls[beg:end]=b` 这种方式修改一个元素或同时修改连续多个元素的值。

但需要注意，在通过 `ls[beg:end]=b` 这种方式赋值时，`b` 是另一个列表，其功能是用 `b` 中各元素替换 `ls` 中 `beg` 至 `end-1` 这些位置上的元素，赋值前后列表元素数量允许发生变化。

元素修改



例如

上面所示的代码中，第3行和第7行都是修改列表ls中某一个元素的值，在为单个元素赋值时，可以使用任意类型的数据（包括列表，如第7行）；第5行是将列表ls中第2至4个元素修改为另一个列表['python',20]中的两个元素；第9行是将列表ls中前两个元素修改为另一个空列表[]中的元素，相当于将ls中前两个元素删除。

Tuple数据类型

概述

01



Tuple（元组）与列表类似，可以包含多个元素，且元素类型可以不相同，书写时每两个元素之间也是用逗号分隔。

02



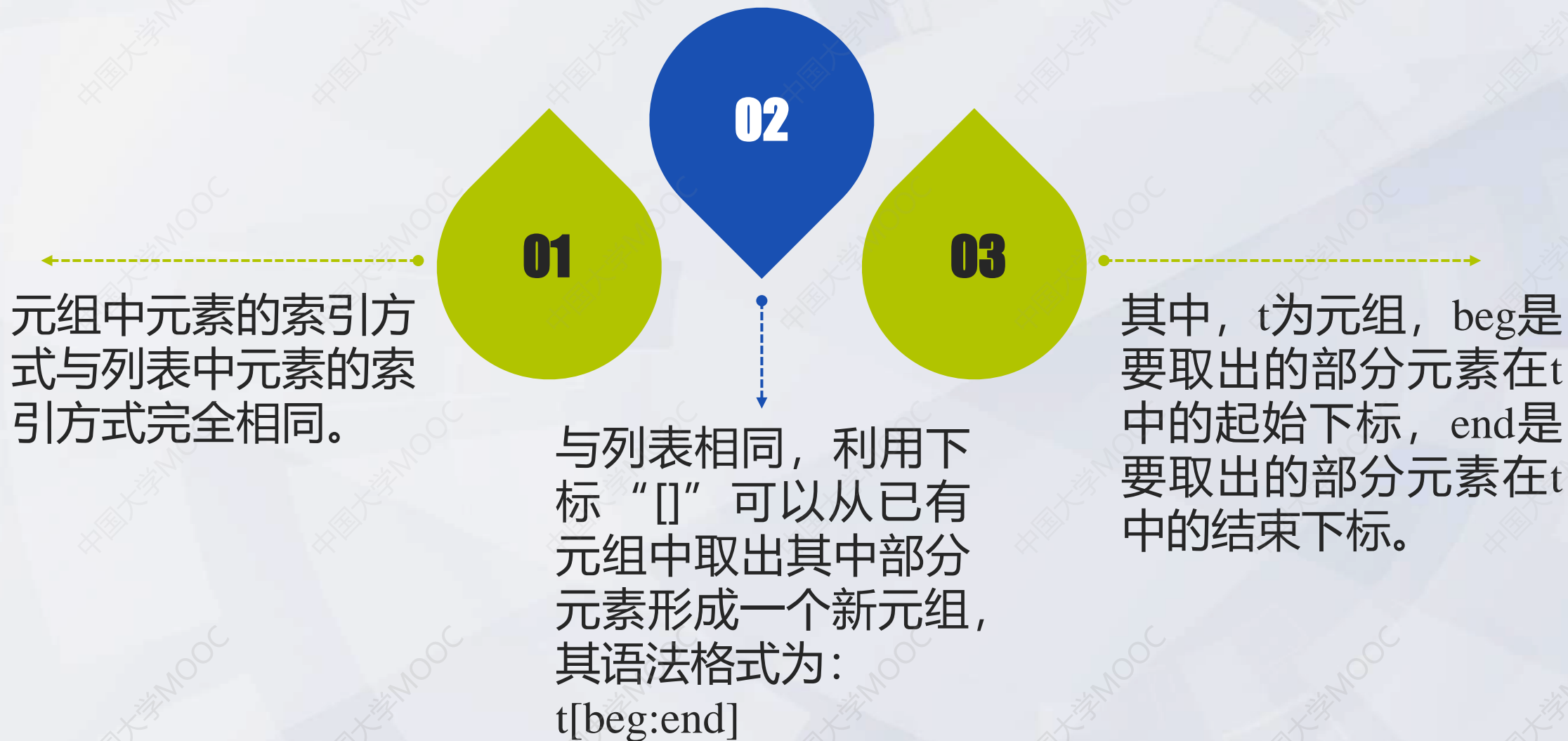
与列表的不同之处在于：元组的所有元素都写在一对小括号“()”中，且元组中的元素不能修改。

03



对于不包含任何元素的元组，即`()`，称为空元组。

元组元素索引



元组元素索引



提示:

省略beg, 则表示从t中的第一个元素开始, 等价于t[0:end];
省略end, 则表示要取出的部分元素从beg位置开始一直到最后一个元素 (包括最后一个元素); beg和end都省略则取出t中的所有元素。

元组元素索引



例如：

1.t=(1, 2.5, 'test', 3+4j, True, [3,1.63], 5.3)

2.print(t[1:4]) #输出 "(2.5, 'test', (3+4j))"

3.print(t[-3:-1]) #输出 "(True, [3, 1.63])"

4.print(t[2:-1]) #输出 "('test', (3+4j), True, [3, 1.63])"

5.print(t[:3]) #输出 "(1, 2.5, 'test')"

6.print(t[-2:]) #输出 "([3, 1.63], 5.3)"

7.print(t[:]) #输出 "(1, 2.5, 'test', (3+4j), True, [3, 1.63], 5.3)"

元组	1	2.5	'test'	3+4j	True	[3,1.63]	5.3
从前向后索引	0	1	2	3	4	5	6
从后向前索引	-7	-6	-5	-4	-3	-2	-1

单一元素访问



如果只访问元组t中的某一个元素，则可以使用下面的写法：

`t[idx]`

其中，idx是要访问的元素的下标。



例如：

1. `t=(1, 2.5, 'test', 3+4j, True, [3,1.63], 5.3)`

2. `print(t[2])` #输出 "test"

3. `print(t[-3])` #输出 "True"

单一元素访问



提示：

字符串、列表和元组的元素都是按下标顺序排列，可通过下标直接访问，这样的数据类型统称为序列。
其中，字符串和元组中的元素不能修改，而列表中的元素可以修改。

Set数据类型

概述

01

与元组和列表类似，Set（集合）中同样可以包含多个不同类型的元素，但集合中的各元素无序、不允许有相同元素且元素必须是可哈希（hashable）的对象。

02

可哈希对象是指拥有 `__hash__(self)` 内置函数的对象。列表、集合和字典类型的数据不是可哈希对象，所以它们不能作为集合中的元素。

创建集合



集合中的所有元素都写在一对大括号 “{}” 中，各元素之间用逗号分隔。创建集合时，既可以使用 {}, 也可以使用 set 函数。set 函数的语法格式如下：set([iterable])

其中，iterable 是一个可选参数，表示一个可迭代对象。



注意：

可迭代 (iterable) 对象是指可以一次返回它的一个元素，如前面学习的字符串、列表、元组都是可迭代的数据类型。

创建集合



例如：

1. `a={ 10, 2.5, 'test', 3+4j, True, 5.3, 2.5}`
2. `print(a)` #输出 `"{True, 2.5, 5.3, 10, (3+4j), 'test'}"`
3. `b=set('hello')`
4. `print(b)` #输出 `"{'e', 'l', 'o', 'h'}"`
5. `c=set([10, 2.5, 'test', 3+4j, True, 5.3, 2.5])`
6. `print(c)` #输出 `"{True, 2.5, 5.3, 10, (3+4j), 'test'}"`
7. `d=set((10, 2.5, 'test', 3+4j, True, 5.3, 2.5))`
8. `print(d)` #输出 `"{True, 2.5, 5.3, 10, (3+4j), 'test'}"`

创建集合



注意：

与字符串、列表、元组等序列类型不同，集合中的元素不能使用下标方式访问。

集合主要用于做并、交、差等集合运算，以及基于集合进行元素的快速检索。

`{}`用于创建空字典，如果要创建一个空集合，则需要使用`set()`。

Dictionary数据类型

概述



01

Dictionary (字典)
是另一种无序的对象集合。



02

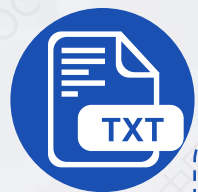
但与集合不同，字典是一种
映射类型，每一个元素是一个键 (key) : 值 (value) 对。

概述



在一个字典对象中

键必须是唯一的，即不同元素的键不能相同；
另外，键必须是可哈希数据，即键不能是列表、集合、字典等类型；
值可以是任意类型。



对于不包含任何元素的字典，即`{}`，称为空字典。

创建字典



既可以使用`{}`，也可以使用`dict`函数。

如果要创建一个空字典，可以使用`{}`或`dict()`。例如：

```
1      a={}
```

```
2      b=dict()
```

执行完毕后，`a`和`b`是两个不包含任何元素的空字典。

创建字典



如果在创建字典的同时，需要给出字典中的元素，则可以使用下面的方法：

1. `{k1:v1,k2:v2,...,kn:vn}` # k_i 和 v_i ($i=1,2,\dots,n$)分别是每一个元素的键和值
2. `dict(**kwarg)` #**kwarg是一个或多个赋值表达式，两个赋值表达式之间用逗号分隔
3. `dict(z)` #z是zip函数返回的结果
4. `dict(ls)` #ls是元组的列表，每个元组包含两个元素，分别对应键和值
5. `dict(dictionary)` #dictionary是一个已有的字典

创建字典



例如：

```
1.a={'one':1, 'two':2, 'three':3}
```

```
2.b=dict(one=1, two=2, three=3)
```

```
3.c=dict(zip(['one','two','three'], [1,2,3]))
```

```
4.d=dict([('one',1), ('two',2), ('three',3)])
```

```
5.e=dict({'one':1, 'two':2, 'three':3})
```



这5条语句创建的5个字典对象的元素完全相同，使用print函数查看每一个变量，都能得到如下输出结果：

```
{'one': 1, 'two': 2, 'three': 3}
```

创建字典

01

zip函数的参数是多个可迭代的对象（列表等），其功能是将不同对象中对应的元素分别打包成元组，然后返回由这些元组组成的列表。

02

在Python 3.x中为了减少内存，zip函数返回的是一个对象，可以通过list函数转换为列表，如通过“`list(zip(['one','two','three'], [1,2,3]))`”可得到列表“`[('one', 1), ('two', 2), ('three', 3)]`”。

访问字典元素



与列表等序列对象不同，在访问字典中的元素时不能通过下标方式访问，而是通过键访问。例如：

```
1.info={'name':'张三', 'age':19, 'score':{'python':95,'math':92}}
```

```
2.print(info['name']) #输出 "张三"
```

```
3.print(info['age']) #输出 "19"
```

```
4.print(info['score']) #输出 "{ 'python': 95, 'math': 92 }"
```

```
5.print(info['score']['python']) #输出 "95"
```

```
6.print(info['score']['math']) #输出 "92"
```

占位运算符和 算术运算符

占位运算符



运算符类似于C语言中sprintf或printf函数中使用的占位符，在字符串中可以给出一些占位符用来表示不同类型的数据，而实际的数据值在字符串之外给出。



占位符	描述	占位符	描述
%d	有符号整型十进制数	%s	字符串
%f或%F	有符号浮点型十进制数		



例如

- `s1='%s上次数学成绩%d, 本次%d, 成绩提高%f' % ('小明',85,90,5/85)`
- `s2='%5s上次数学成绩%5d, 本次%5d, 成绩提高%.2f' % ('小明',85,90,5/85)`
- `s3='%5s上次数学成绩%05d, 本次%05d, 成绩提高%08.2f' % ('小明',85,90,5/85)`



执行完毕后，通过print函数分别输出s1、s2和s3，可得到下面结果：

- 小明上次数学成绩85，本次90，成绩提高0.058824
- 小明上次数学成绩 85，本次 90，成绩提高0.06
- 小明上次数学成绩00085，本次00090，成绩提高00000.06

提示

由于%作为占位符的前缀字符，因此对于有占位符的字符串，表示一个%时需要写成“%%”。例如，执行“`print('优秀比例为%.2f%%，良好比例为%.2f%%。'%(5.2,20.35))`”，输出结果为：优秀比例为5.20%，良好比例为20.35%。

算术运算符



算术运算是计算机支持的主要运算之一，其运算对象是数值型数据。

运算符	使用方法	功能描述
+ (加)	$x+y$	x与y相加
- (减)	$x-y$	x与y相减
* (乘)	$x*y$	x与y相乘
/ (除)	x/y	x除以y
// (整除)	$x//y$	x整除y，返回x/y的整数部分
% (模)	$x\%y$	x整除y的余数，即 $x-x//y*y$ 的值
- (负号)	$-x$	x的负数
+ (正号)	$+x$	c
** (乘方)	$x**y$	x的y次幂

例如

```
1. i1,i2=10,3
2. f1,f2=3.2,1.5
3. c1,c2=3+4.1j,5.2+6.3j
4. print(i1+i2) #输出 "13"
5. print(c1-c2) #输出 "(-2.2-2.2j) "
6. print(f1*f2) #输出
   "4.8000000000000001"
7. print(i1/i2) #输出
   "3.3333333333333335"
8. print(i1//i2) #输出 "3"
9. print(i1%i2) #输出 "1"
10. print(-f1) #输出 "-3.2"
11. print(+f2) #输出 "1.5"
12. print(i1**i2) #输出 "1000"
```



提示

十进制小数在转换为二进制时有可能产生精度损失，所以在第6行和第7行的输出中，结果与实际计算结果之间存在偏差，如f1 (3.2) 乘以f2 (1.5) 应该等于4.8，但最后输出的数据与实际计算结果存在0.0000000000000001的偏差。

赋值运算符 比较运算符 和逻辑运算符

赋值运算符



赋值运算要求左操作数对象必须是值可以修改的变量。

运算符	使用方法	功能描述
=	$y=x$	将x的值赋给变量y
+=	$y+=x$	等价于 $y=y+x$
-=	$y-=x$	等价于 $y=y-x$
=	$y=x$	等价于 $y=y*x$
/=	$y/=x$	等价于 $y=y/x$
//=	$y//=x$	等价于 $y=y//x$
%=	$y\%=x$	等价于 $y=y\%x$
=	$y=x$	等价于 $y=y**x$



例如

1. `i1,i2=10,3` #i1和i2的值分别被赋为10和3
2. `i1+=i2` #i1的值被改为13
3. `print(i1)` #输出 "13"
4. `c1,c2=3+4.1j,5.2+6.3j` #c1和c2的值分别被赋为 $3+4.1j$ 和 $5.2+6.3j$
5. `c1-=c2` #c1的值被改为 $-2.2-2.2j$
6. `print(c1)` #输出 " $-2.2-2.2j$ "
7. `f1,f2=3.2,1.5` #f1和f2的值分别被赋为3.2和1.5
8. `f1*=f2` #f1的值被改为4.8
9. `print(f1)` #输出 "4.8"
10. `i1,f1=3,0.5` #i1和f1的值分别被赋为3和0.5
11. `i1**=f1` #i1的值被改为1.7320508075688772 (即3的0.5次幂)
12. `print(i1)` #输出 "1.7320508075688772"

比较运算符

▶ 比较运算的作用是对两个操作数对象的大小关系进行判断。

运算符	使用方法	功能描述
== (等于)	y==x	如果y和x相等，则返回True； 否则，返回False
!= (不等于)	y!=x	如果y和x不相等，则返回 True；否则，返回False
> (大于)	y>x	如果y大于x，则返回True； 否则，返回False
< (小于)	y<x	如果y小于x，则返回True； 否则，返回False
>= (大于等于)	y>=x	如果y大于或等于x，则返回 True；否则，返回False
<= (小于等于)	y<=x	如果y小于或等于x，则返回 True；否则，返回False

例如

1. `i1,i2,i3=25,35,25` #i1、i2和i3分别被赋为25、35和60
2. `print(i1==i2)` #输出 "False"
3. `print(i1!=i2)` #输出 "True"
4. `print(i1>i3)` #输出 "False"
5. `print(i1<i2)` #输出 "True"
6. `print(i1>=i3)` #输出 "True"
7. `print(i1<=i2)` #输出 "True"

提示

比较运算返回的结果是布尔值True或False。在执行程序时，程序中的每条语句并不一定是按顺序依次执行。比较运算的主要作用是设置条件，某些语句在满足条件时才会执行一次（即条件语句），而某些语句在满足条件时会重复执行多次（即循环语句）。

逻辑运算符



逻辑运算可以将多个比较运算连接起来形成更复杂的条件判断。

运算符	使用方法	功能描述
and	x and y	如果x和y都为True,则返回True;否则, 返回False
or	x or y	如果x和y都为False,则返回False;否则, 返回True
not	not x	如果x为True,则返回False; 如果x为False, 返回True



例如：

1. `n=80,a=100`
2. `print(n>=0 and n<=a)` #输出 “True” , 判断n是否大于等于0且小于等于a
3. `print(n<0 or n>a)` #输出 “False ” , 判断n是否小于0或大于a
4. `print(not(n>=0 and n<=a))` #输出 “False ”

位运算符

十进制转二进制



除基取余法

用2去除十进制整数，得到商和余数；

如果商不为0，则继续用2除，再得到商和余数，重复该步骤直至商为0；

最后将余数按照从后至前的顺序排列，即得到转换后的二进制数。

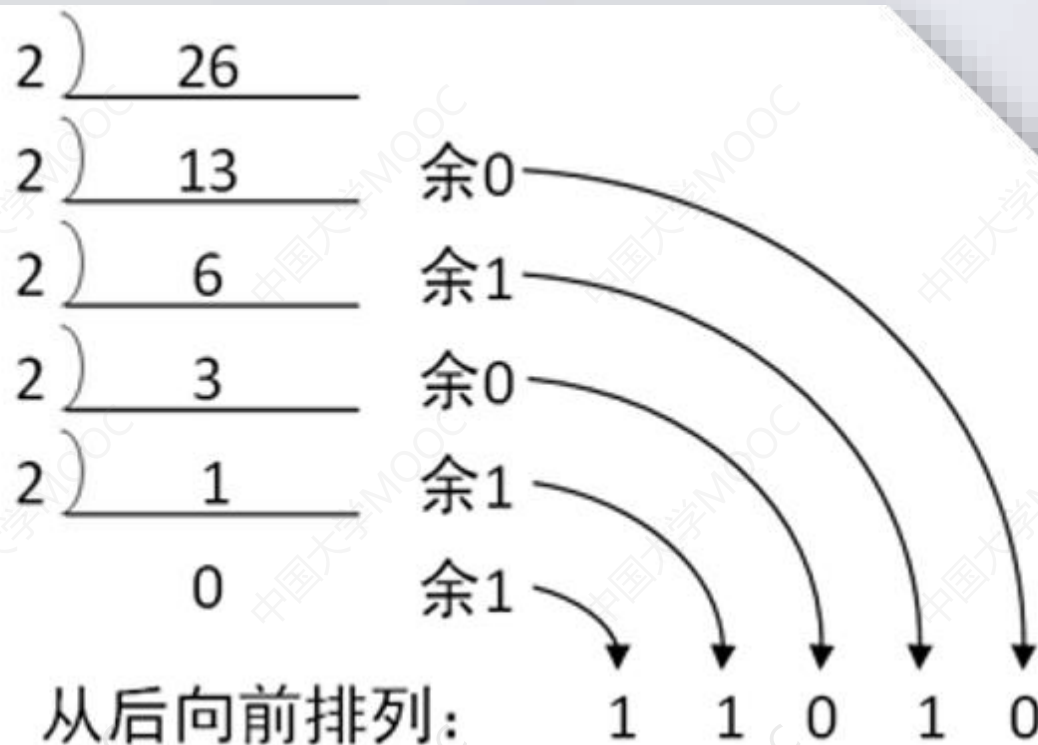


提示

“除基取余法”中的“基”是指基数，基数即为一种数制中可用数码的个数。二进制可用的数码只有0和1两个，所以二进制的基数是2。

十进制转二进制

例如:



二进制转十进制

二进制数转十进制数的规则是“按权展开求和”，即将二进制数的每一位写成数码乘以位权的形式，再对乘积求和。

例如

对于二进制数11010B，其对应的十进制数为：

$$\begin{aligned} 11010B &= 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 \\ &= 1*16 + 1*8 + 0*4 + 1*2 + 0*1 \\ &= 16 + 8 + 0 + 2 + 0 \\ &= 26 \end{aligned}$$

位运算符

位运算是指对二进制数进行逐位运算。

运算符	使用方法	功能描述
& (按位与)	$y \& x$	如果y和x对应位都为1，则结果中该位为1；否则，该位为0
(按位或)	$y x$	如果y和x对应位都为0，则结果中该位为0；否则，该位为1
^ (按位异或)	$y \wedge x$	如果y和x对应位不同，则结果中该位为1；否则，该位为0
<< (左移位)	$y \ll x$	将y左移x位 (右侧补0)
>> (右移位)	$y \gg x$	将y右移x位 (左侧补0)
~ (按位取反)	$\sim x$	如果x的某位为1，则结果中该位为0；否则，该位为1

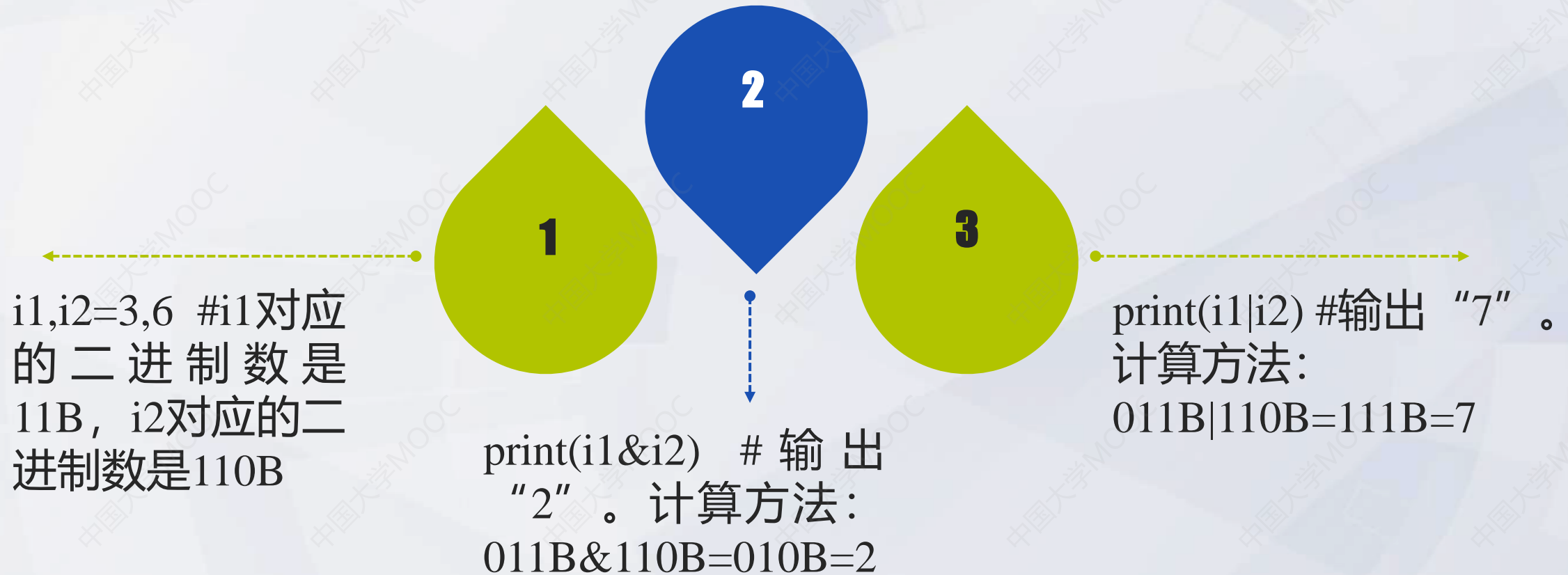
位运算示例

011B
& 110B

010B

位运算示例

例如：



位运算示例

4

▶ `print(i1^i2)` #输出 "5" 。计算方法: $011\text{B} \wedge 110\text{B} = 101\text{B} = 5$

5

▶ `print(i1<<1)` #输出 "6" 。计算方法: $11\text{B} \ll 1 = 110\text{B} = 6$

6

▶ `print(i1>>1)` #输出 "1" 。计算方法: $11\text{B} \gg 1 = 1\text{B} = 1$

身份运算符和成员运算符

身份运算符



身份运算用于比较两个对象是否对应同样的存储单元。

运算符	使用方法	功能描述
is	x is y	如果x和y对应同样的存储单元，则返回True；否则，返回False
is not	x is not y	如果x和y不对应同样的存储单元，则返回True；否则，返回False

提示

程序在运行时，输入数据和输出数据都是存放在内存中。内存中的一个存储单元可以存储一个字节的数 据，每个存储单元都有一个唯一的编号，称为内存地址。根据数据类型不同，其所占用的内存大小也不同。一个数据通常会占据内存中连续多个存储单元，起始存储单元的地址称为该数据的内存首地址。利用id函数可以查看一个数据的内存首地址。

x is y等价于 $\text{id}(x) == \text{id}(y)$ ，即判断x和y的内存首地址是否相同；x is not y等价于 $\text{id}(x) != \text{id}(y)$ ，即判断x和y的内存首地址是否不相同。

身份运算符



例如：

```
1 x,y=15,15
2 print(x is y) #输出 "True"
3 print(x is not y) #输出 "False"
4 print(x is 15) #输出 "True"
5 x,y=[1,2,3],[1,2,3]
6 print(x is y) #输出 "False"
7 print(x==y) #输出 "True"
8 print(x is [1,2,3]) #输出 "False"
9 x=y
10 print(x is y) #输出 "True"
```



提示：

如果赋值运算符 “=” 的右操作数也是一个变量，则赋值运算后左操作数变量和右操作数变量会对应同样的存储单元。

成员运算符

成员运算符

成员运算用于判断一个可迭代对象（序列、集合或字典）中是否包含某个元素。

运算符	使用方法	功能描述
in	x in y	如果x是可迭代对象y的一个元素，则返回True；否则，返回False
not in	x not in y	如果x不是可迭代对象y的一个元素，则返回True；否则，返回False

成员运算符



例如：

```
1 x,y=15,['abc',15,True]
2 print(x in y) #输出 "True"
3 x=20
4 print(x not in y) #输出 "True "
5 y=(20,'Python')
6 print(x in y) #输出 "True "
7 x,y='Py','Python'
8 print(x in y) #输出 "True "
9 x,y=20,{ 15,20,25}
10 print(x in y) #输出 "True "
11 x,y='one',{'one':1,'two':2,'three':3}
12 print(x in y) #输出 "True "
13 print(1 in y) #输出 "False "
```



提示：

使用成员运算符判断一个数据是否是字典中的元素，实际上就是判断该数据是否是字典中某个元素的键。

序列运算符和运算符优先级

序列运算符



用于序列操作的运算符

运算符	使用方法	功能描述
+ (拼接)	$x+y$	将序列x和序列y中的元素连接，生成一个新的序列
* (重复)	$x*n$	将序列x中的元素重复n次，生成一个新的序列

序列运算符



例如：

```
1 x,y=[12,False],['abc',15,True]
2 z=x+y #x和y拼接后的结果赋给z
3 print(z) #输出 "[12, False, 'abc', 15, True]"
4 s1,s2='我喜欢学习','Python'
5 s=s1+s2 #s1和s2拼接后的结果赋给s
6 print(s) #输出 "我喜欢学习Python"
7 x_3=x*3 #将序列x的元素重复3次，生成一个新序列并赋给x_3
8 print(x_3) #输出 "[12, False, 12, False, 12, False] "
9 s_3=s*3 #将字符串s重复3次，生成一个新字符串并赋给s_3
10 print(s_3) #输出 "我喜欢学习Python我喜欢学习Python我喜欢学习Python"
```

运算符优先级

在一个表达式中，通常会包含多个运算，这就涉及到了运算的顺序，其由两个因素确定：运算符的优先级和运算符的结合性。



优先级

- 对于具有不同优先级的运算符，会先完成高优先级的运算，再完成低优先级的运算。
- 例如，表达式 $3+5*6$ 中，“ $*$ ”优先级高于“ $+$ ”，因此会先计算 $5*6$ ，再计算 $3+30$ 。

结合性



对于具有相同优先级的运算符，其运算顺序由结合性来决定。结合性包括左结合和右结合两种，左结合是按照从左向右的顺序完成计算，而右结合是按照从右向左的顺序完成计算。

例如，表达式 $5-3+6$ 中，“ $-$ ”和“ $+$ ”优先级相同，它们是左结合的运算符，因此会先计算 $5-3$ ，再计算 $2+6$ ；表达式 $a=b=1$ 中，“ $=$ ”是右结合的运算符，因此会先计算 $b=1$ ，再计算 $a=b$ 。

运算符优先级

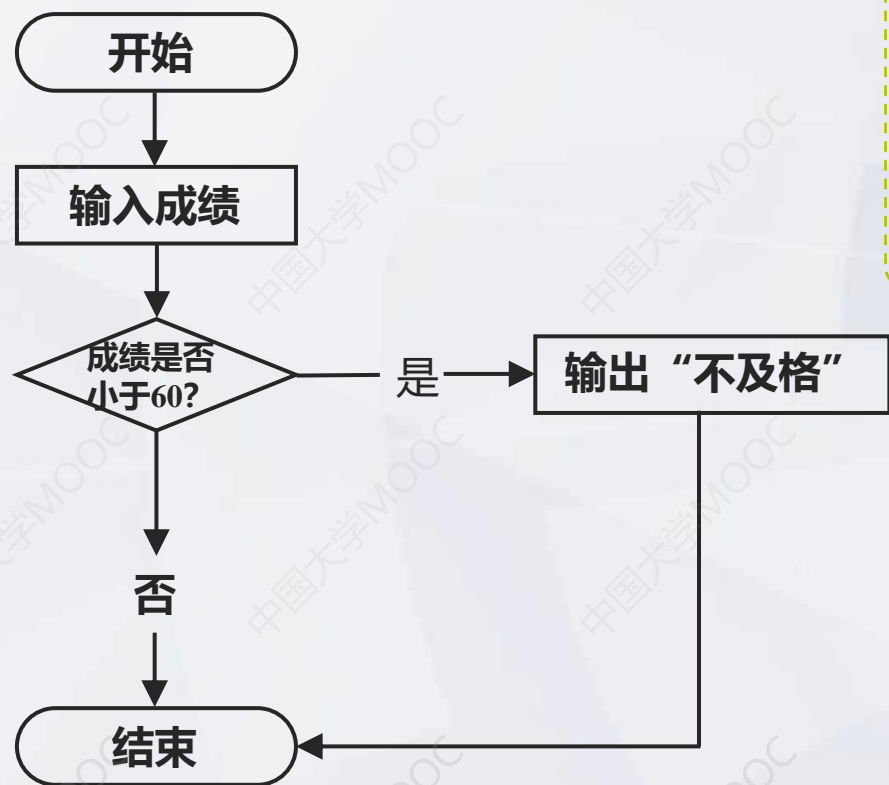
优先级	运算符	描述
1	**	乘方
2	~、+、-	按位取反、正号、负号
3	*, /、//、%	乘/序列重复、除、整除、模
4	+, -	加/序列连接、减
5	>>、<<	右移位、左移位
6	&	按位与
7	^	按位异或
8		按位或
9	>、<、>=、<=、==、!=、is、is not、in、not in	比较运算符、身份运算符、成员运算符
10	=、+=、-=、*=、/=、//=、%=、**=	赋值运算符
11	not	逻辑非
12	and	逻辑与
13	or	逻辑或

条件语句概述

条件语句概述



通过设置条件，可以使得某些语句在条件满足时才会执行。



- 1 输入成绩并保存到变量score中
- 2 如果score小于60
- 3 输出 “不及格”

提示

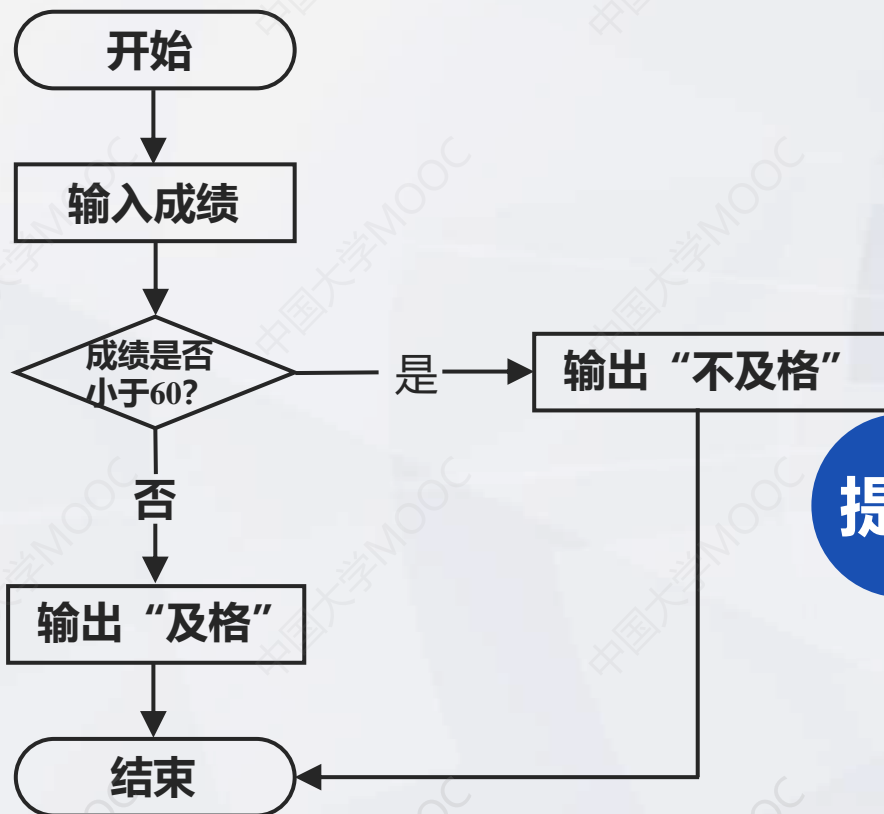
在解决一个实际问题时，可以先使用流程图、自然语言或伪代码等形式描述数据处理流程（即算法设计），再按照设计好的流程（即算法）编写程序。

这样，在设计算法时可以忽略具体代码实现、而专注于如何解决问题，有利于避免程序的逻辑错误。

条件语句概述



通过设置条件，可以使得某些语句在条件满足时才会执行。

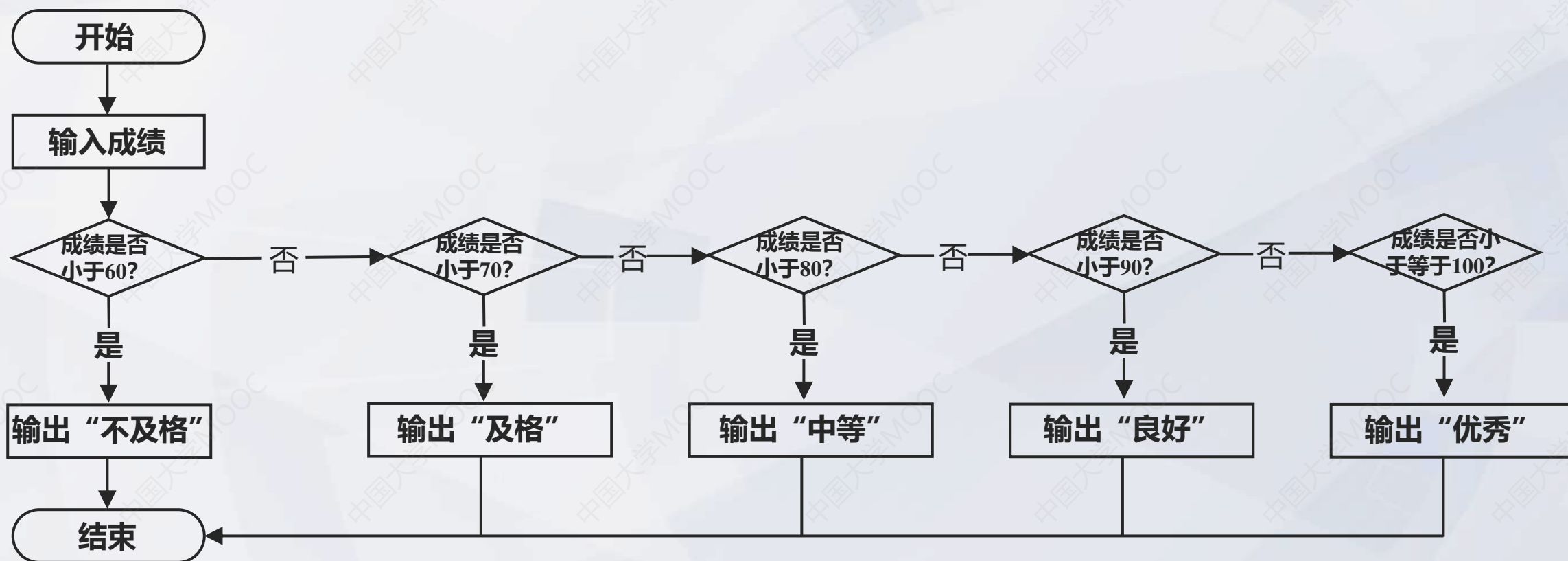


- 1 输入成绩并保存到变量score中
- 2 如果score小于60
- 3 输出“不及格”
- 4 否则
- 5 输出“及格”

提示

在绘制流程图时，要求必须从“开始”出发，经过任何处理后必然能到达“结束”。另外，流程图中使用的图形符号有着严格规定，“开始”和“结束”一般放在圆角矩形或圆中，数据处理放在矩形框中，而条件判断放在菱形框中。

条件语句概述



条件语句概述



例如：

- 1 输入成绩并保存到变量score中
- 2 如果score小于60
- 3 输出 “不及格”
- 4 否则，如果score小于70
- 5 输出 “及格”
- 6 否则，如果score小于80
- 7 输出 “中等”
- 8 否则，如果score小于90
- 9 输出 “良好”
- 10 否则，如果score小于等于100 #显然，可以将条件去掉，直接改为 “否则”
- 11 输出 “优秀”

条件语句实现和pass

条件语句语法格式



if、elif、else

```
if 条件1:  
    语句序列1  
[elif 条件2:  
    语句序列2  
.....  
elif 条件K:  
    语句序列K]  
[else:  
    语句序列K+1]
```

提示:

if表示“如果”，elif表示“否则如果”，else表示“否则”。最简单的条件语句只有if，elif和else都是可选项，根据需要决定是否使用。

条件语句示例



例如：

```
1 score=eval(input('请输入成绩 (0~100之间的整数) : '))
2 if score<60: #注意要写上 ":"
3     print('不及格')
```

```
1 score=eval(input('请输入成绩 (0~100之间的整数) : '))
2 if score<60:
3     print('不及格')
4 else: #注意else后也要写上 ":"
5     print('及格')
```


条件语句示例



例如：

```
1 score=eval(input('请输入成绩 (0~100之间的整数) : '))
2 if score<60:
3     print('不及格')
4 elif score<70: #注意elif后也要写上 ":"
5     print('及格')
6 elif score<80:
7     print('中等')
8 elif score<90:
9     print('良好')
10 elif score<=100: #也可以改为 "else:"
11     print('优秀')
```

条件语句示例

每一个语句序列中可以包含一条或多条语句。例如：



```
1      score=eval(input('请  
输入成绩 (0~100之间的整  
数) : '))  
2      if score<60:  
3          print('你的成绩是  
%d'%score)  
4          print('不及格')
```



```
1      score=eval(input('请输入成  
绩 (0~100之间的整数) : '))  
2      if score<60:  
3          print('你的成绩是  
%d'%score)  
4          print('不及格') #缺少缩进
```

pass

pass表示一个空操作，只起到一个占位作用，执行时什么都不做。



```
1 score=eval(input('请输入成绩（0~100之间的整数）：'))
2 if score>=60:
3     pass #什么都不做
4 else:
5     print('不及格')
```

pass

pass表示一个空操作，只起到一个占位作用，执行时什么都不做。

提示：

在某些必要的语句（如条件语句中的各语句序列）还没有编写的情况下，如果要运行程序，则可以先在这些必要语句处写上“pass”，使得程序不存在语法错误、能够正常运行。

实际上，pass与条件语句并没有直接关系，在程序中所有需要的地方都可以使用pass作为占位符。比如，在后面将要学习的循环语句中，也可以使用pass作为占位符。

循环语句概述和for循环

概述

通过循环，可以使得某些语句重复执行多次。

例如

我们要计算从1到n的和，可以使用一个变量 $sum=0$ 保存求和结果，并设置一个变量 i 、让其遍历1到n这n个整数；对于 i 的每一个取值，执行 $sum+=i$ 的运算；遍历结束后， sum 中即保存了求和结果。

概述



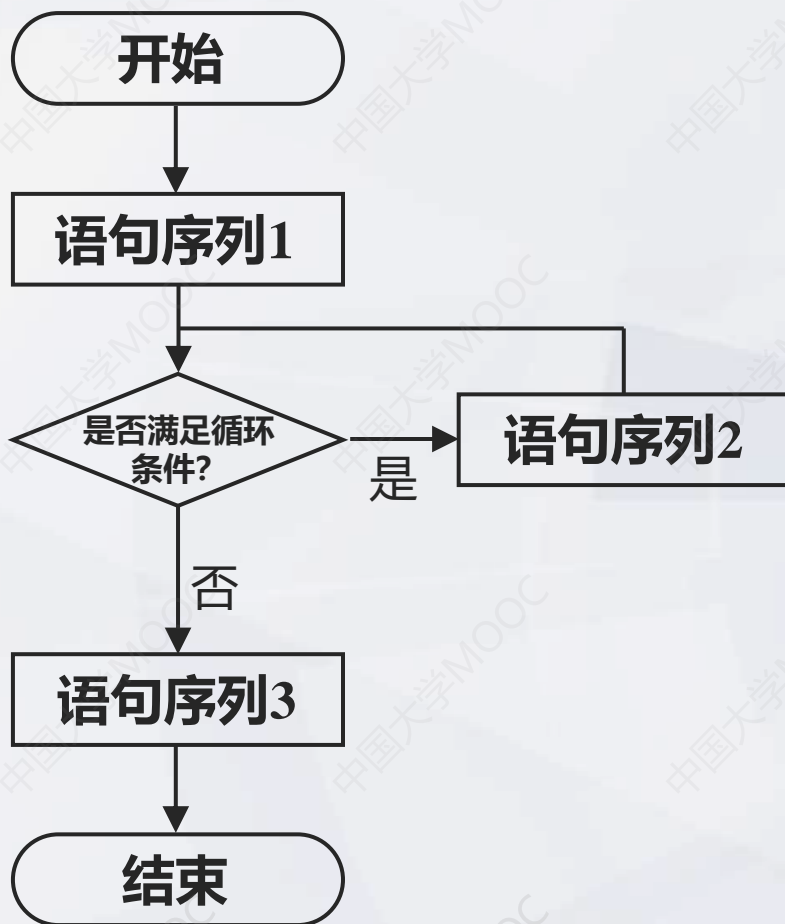
提示

“遍历”这个词在计算机程序设计中经常会用到，其表示对某一个数据中的数据元素按照某种顺序进行访问，使得每个数据元素访问且仅访问一次。

例如

▶ 对于列表`ls=[1, 'Python', True]`中的3个元素，如果按照某种规则（如从前向后或从后向前）依次访问了1、'Python'、True这3个元素，且每个元素仅访问了一次，则可以说对列表`ls`完成了一次遍历。

循环语句执行过程



提示

循环条件判断和语句序列2构成了循环语句：只要满足循环条件，就会执行语句序列2；执行语句序列2后，会再次判断是否满足循环条件。

for循环

用于遍历可迭代对象中的每一个元素，并根据当前访问的元素做数据处理，其语法格式为：

for 变量名 in 可迭代对象:
语句序列

例如

```
1 ls=['Python','C++','Java']  
2 for k in ls:  
3     print(k)
```

Python
C++
Java

for循环

再如

```
1 d={'Python':1,'C++':2,'Java':3}
2 for k in d: #注意for后要写上 ":"
3     print('%s:%d'%(k,d[k]))
```

Python:1

C++:2

Java:3

提示

使用for遍历字典中的元素时，每次获取到的是元素的键，通过键可以再获取到元素的值。

for循环

使用for循环时，如果需要遍历一个数列中的所有数字，则通常利用range函数生成一个可迭代对象。

range函数的语法格式如下：

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

for循环

例如



```
1 print(list(range(1,5,2))) #输出 "[1, 3]"
2 print(list(range(5,-1,-2))) #输出 "[5, 3, 1]"
3 print(list(range(1,5))) #输出 "[1, 2, 3, 4]"
4 print(list(range(5))) #输出 "[0, 1, 2, 3, 4]"
```

提示

range函数返回的是一个可迭代对象，通过list函数可将该对象转换为列表。

for循环

例



使用for循环实现1到n的求和。

```
1 n=eval(input('请输入一个大于0的整数: '))
2 sum=0
3 for i in range(1,n+1): #range函数将生成由1到n这n个整数组成的可迭代对象
4     sum+=i
5 print(sum) #输出求和结果
```

for循环

例

► 使用for循环实现1到n之间所有奇数的和。

```
1 n=eval(input('请输入一个大于0的整数: '))
2 sum=0
3 for i in range(1,n+1,2): #步长2, 因此会生成1、3、5、...等奇数
4     sum+=i
5 print(sum) #输出求和结果
```


while 循环和索引

while循环



语法格式

while 循环条件:
语句序列

例

► 使用while循环实现1到n的求和。

```
1 n=eval(input('请输入一个大于0的整数: '))
2 i,sum=1,0 #i和sum分别赋值为1和0
3 while i<=n: #当i<=n成立时则继续循环, 否则退出循环
4     sum+=i
5     i+=1 #注意该行也是while循环语句序列中的代码, 与第4行代码应有相同缩进
6 print(sum) #输出求和结果
```

while循环

例

▶ 使用while循环实现1到n之间所有奇数的和。

```
1 n=eval(input('请输入一个大于0的整数: '))
2 i,sum=1,0
3 while i<=n:
4     sum+=i
5     i+=2
6 print(sum) #输出求和结果
```

索引

如果希望不仅获取到每一个元素的值，而且能获取到每一个元素的索引，则可以通过len函数获取可迭代对象中的元素数量，再通过range函数生成由所有元素索引组成的可迭代对象。

例

▶ 同时访问索引和元素值。

```
1 ls=['Python','C++','Java']  
2 for k in range(len(ls)): #k为每一个元素的索引  
3     print(k,ls[k]) #通过ls[k]可访问索引为k的元素
```

```
0 Python  
1 C++  
2 Java
```

索引

也可以利用`enumerate`函数返回的索引序列对象同时获得每个元素的索引和值。

例

► 利用`enumerate`函数访问索引和元素值。

```
1 ls=['Python','C++','Java']  
2 for k,v in enumerate(ls): #k保存当前元素索引, v保存当前元素值  
3     print(k,v)
```

0 Python
1 C++
2 Java

```
1 ls=['Python','C++','Java']  
2 for k,v in enumerate(ls,1): #索引从1开始 (默认为0)  
3     print(k,v)
```

1 Python
2 C++
3 Java

break、continue和else

break

用于跳出for循环或while循环。对于多重循环情况，跳出最近的那重循环。

例

► 求1~100之间的素数。

```
1 for n in range(2,101): #n在2~100之间取值
2     m=int(n**0.5) #m等于根号n取整
3     i=2
4     while i<=m:
5         if n%i==0: #如果n能够被i整除
6             break #跳出while循环
7         i+=1
8     if i>m: #如果i>m, 则说明对于i从2到m上的取值、都不能整除n, 所以n是素数
9         print(n,end=' ') #输出n
```


break



输出结果

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53  
59 61 67 71 73 79 83 89 97
```

continue

用于结束本次循环并开始下一次循环。对于多重循环情况，作用于最近的那重循环。

例

▶ 3的倍数的整数求和。

```
1 sum=0
2 while True: #因为循环条件设置为True，所以无法通过条件不成立退出循环
    #（永真循环）
3     n=eval(input('请输入一个整数（输入0结束程序）：'))
4     if n==0: #如果输入的整数是0，则通过break跳出循环
5         break
6     if n%3!=0: #如果n不是3的倍数，则不做求和运算
7         continue #通过continue结束本次循环、开始下一次循环，即转到第2
    #行代码
8     sum+=n #将n加到sum中
9 print('所有是3的倍数的整数之和为： %d'%sum)
```

continue



执行程序时

依次输入10、15、20、25、30、0，则最后输出45
(即 $15+30$ 的结果)。

else

在for循环和while循环后面可以跟着else分支，当for循环已经遍历完列表中所有元素或while循环的条件为False时，就会执行else分支。

例

▶ 素数判断

```
1 n=eval(input('请输入一个大于1的整数: '))
2 m=int(n**0.5) #m等于根号n取整
3 for i in range(2,m+1): #i在2至m之间取值
4     if n%i==0: #如果n能够被i整除
5         break #跳出while循环
6 else: #注意这个else与第3行的for具有相同的缩进，所以它们是同一层次的语句
7     print('%d是素数'%n)
```

else



执行程序时

如果输入5，则会输出“5是素数”；
如果输入10，则不会输出任何信息。