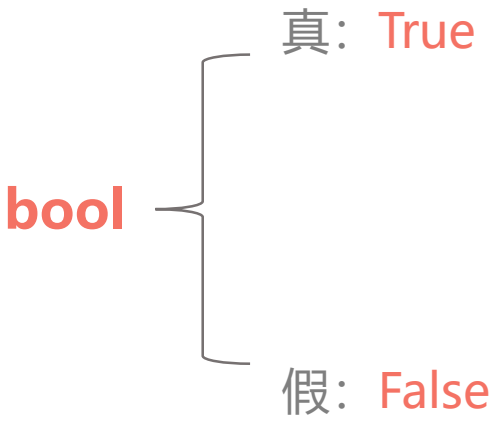
布尔值

用来表示条件是否成立。

注意：非零非空的数值可以表示真。



比较运算符

比较运算符：用于判断条件是否成立。

操作符	操作含义
<	小于
<=	小于等于
>=	大于等于
>	大于
==	等于
!=	不等于

逻辑运算符

逻辑运算符：用于检测两个或两个以上的条件是否满足 。

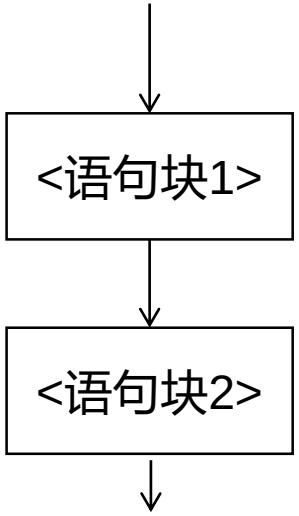
操作符	操作含义	描述
and	逻辑“与”	当两边都为真，结果为真
or	逻辑“或”	只要有一边结果为真，结果为真
not	逻辑“非”	取反结果

程序的三种控制结构

程序由3种基本结构组成：顺序结构、分支结构和循环结构。

任何程序都由这3种基本结构组合而成。

顺序结构是程序按照线性顺序依次执行的一种运行方式。如左图所示。



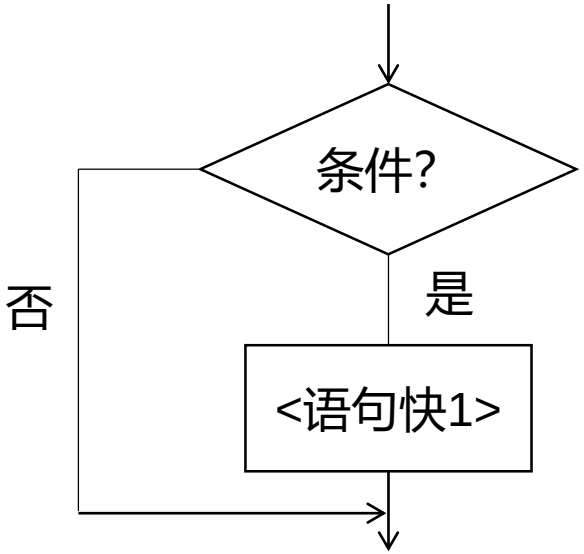
print('hello world')

print('世界，你好！')

单分支结构

分支结构是程序根据条件判断结果而选择不同向前执行路径的一种运行方式。

单分支控制过程的流程图如下图所示。



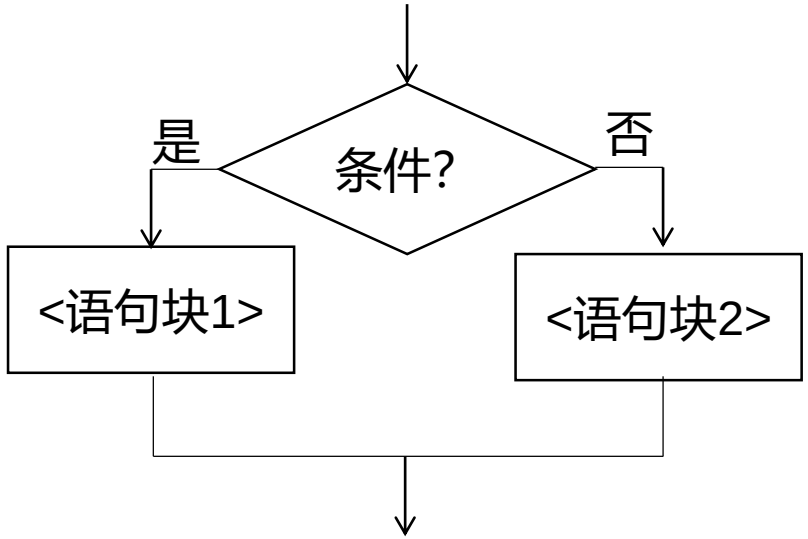
语法格式：

```
if <条件>:
    <语句块>
```

案例

输入学生分数，如果大于60分，则提醒学生考试通过，并输出具体的分数。

二分支结构



语句1是在条件为真时执行，语句2是在条件为假时执行。

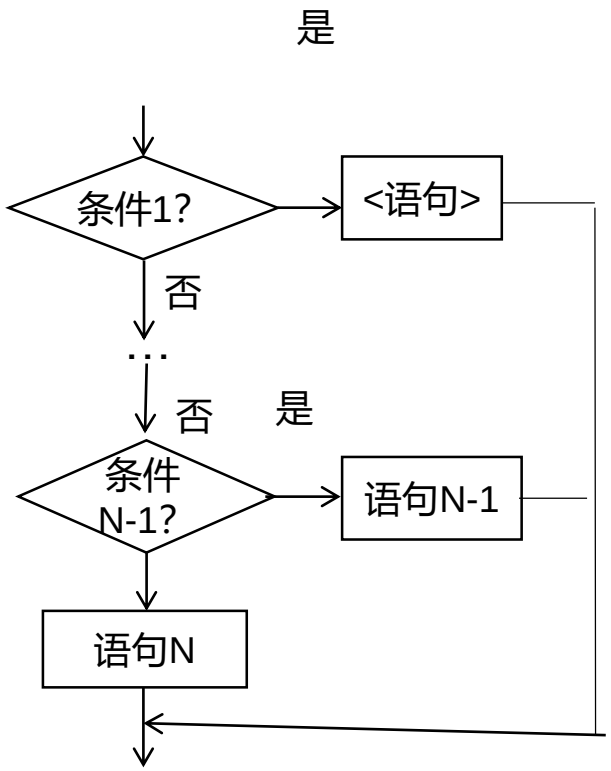
语法格式：

```
if  <条件>:
      <语句块1>
else:
      <语句块2>
```

案例

输入学生分数，如果大于60分，则提醒学生考试通过，否则提醒学生考试不及格。

多分支结构



多分支结构一般用于判断同一个条件或一类条件的多个执行路径。依次判断条件并执行第一个条件为True的语句。若没有条件成立，执行else下面的语句。

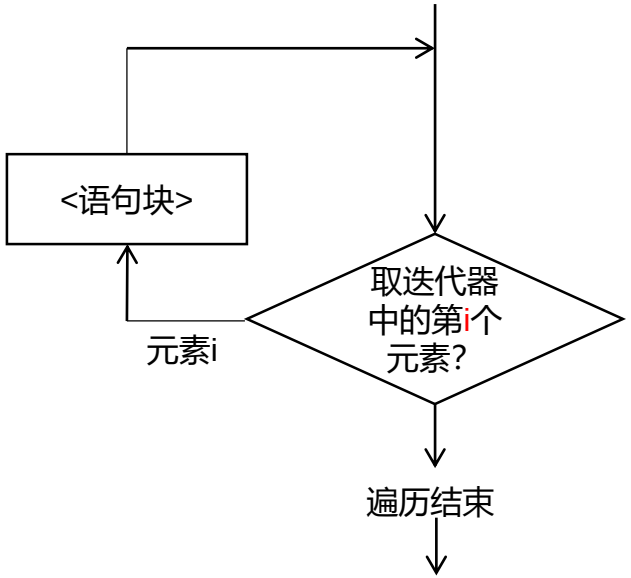
语法格式：

```
if   <条件1>:
        <语句块1>
elif  <条件2>:
        <语句块2>
elif <条件3>:
        <语句块3>
…
else:
      <语句块N>
```

案例

输入学生分数，将学生分数转换为等级。
90-100--->A
80-90 --->B
70-80 --->C
60-70 --->D
0-60 --->E

for循环



从数据集合中逐一提取元素，放在循环变量中，对于每个提取的元素执行一次语句块。for语句的循环执行次数是根据集合中元素个数确定的。

语法格式：

```
for 变量名 in 数据集合:
    <语句>
```

数据集合可以是字符串、文件、range()函数或其他组合数据类型等。

遍历方式

对于字符串，可以逐一遍历字符串的每个字符，

基本使用方式如下：

```
for <循环变量> in <字符串变量>:
    <语句块>
```

```
>>> for i in "python":
        print(i)

p
y
t
h
o
n
```

遍历方式

for循环经常和range()函数一起使用，

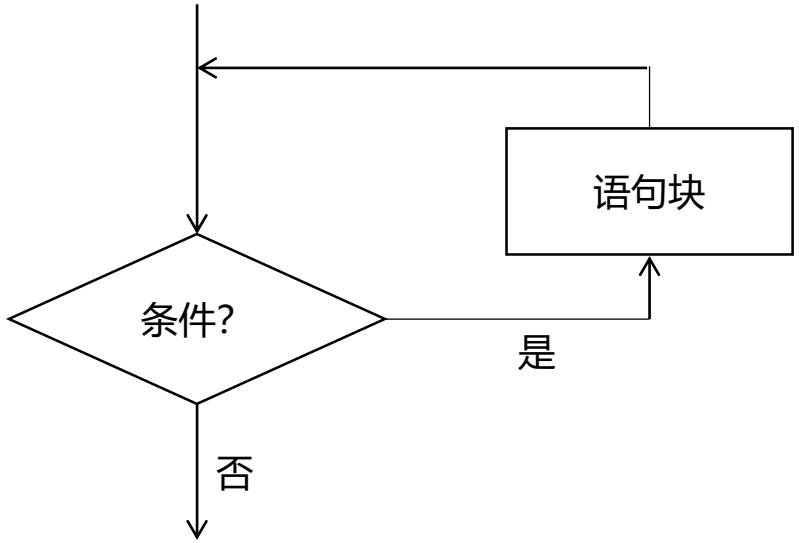
基本使用方式如下：

```
for <循环变量> in range(<循环次数>):
    <语句块>
```

```
>>> for i in range(1,6):
        print(i)

1
2
3
4
5
```

while循环



程序执行while语句时，判断条件，若为True，执行循环体语句，语句结束后返回再次判断条件；直到条件为False时，循环终止，执行与while同级别的后续语句。

语法格式：

```
while <条件>:
    <代码块>
```

实例：

```
n = 0
while n < 10:
    print(n)
    n = n + 3
```

break和continue

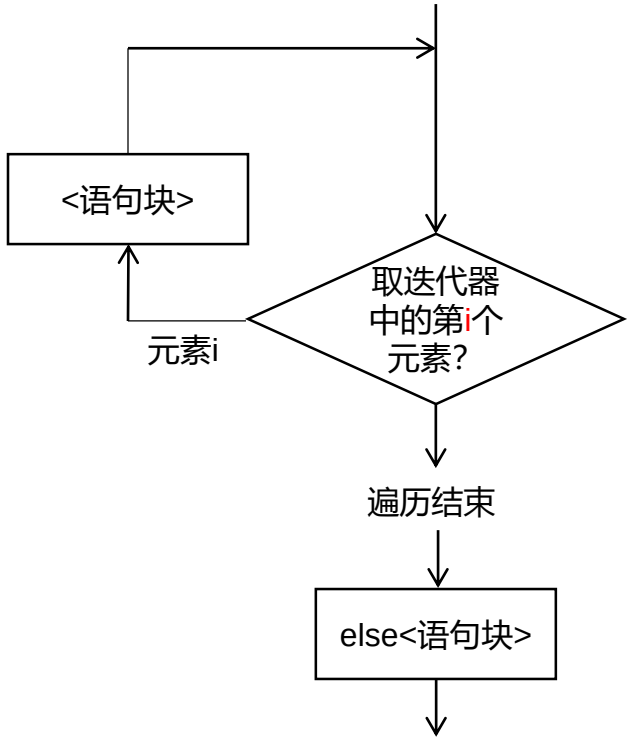
1.continue语句

用来结束当前当次循环，即跳出循环体中下面尚未执行的语句，但不跳出当前循环。

2.break语句

跳出最内层循环，终止该循环后，从循环后的代码继续执行。

for循环的扩展模式



当for循环正常执行之后，程序会继续执行else语句中内容。else语句只在循环正常执行之后才执行并结束。

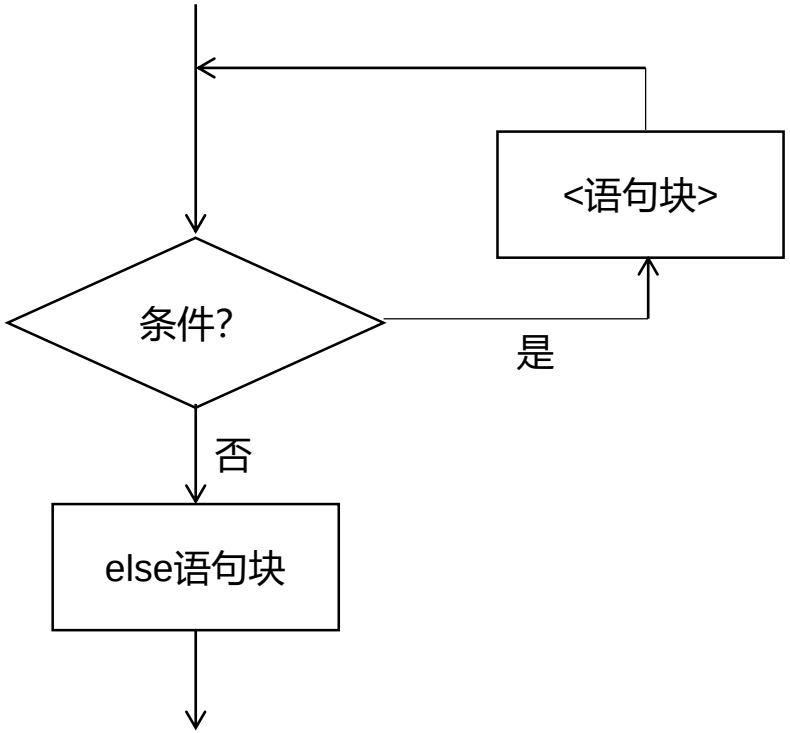
语法格式：

```
for 变量名 in 集合:
    <语句块1>
else:
    <语句块2>
```

实例：

```
for i in 'PY':
    print('循环执行中：'+i)
else:
    i = "循环正常结束"
print(i)
```

while循环的扩展模式



当while循环正常执行之后，程序会继续执行else语句中内容。else语句只在循环正常执行之后才执行并结束。

语法格式：

```
while <条件>:
    <语句块1>
else:
    <语句块2>
```

实例：

```
n = 0
while n<10:
    n = n+3
    print(n)
else:
    print("循环正常结束")
```

异常处理

Python程序一般对输入有一定要求，但当实际输入不满足程序要求时，可能会产生程序的运行错误。所以为了保证程序的稳定运行我们要对错误捕获并合理控制。

```
>>> eval(input("请输入一个数字:"))
请输入一个数字:python
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    eval(input("请输入一个数字:"))
  File "<string>", line 1, in <module>
NameError: name 'python' is not defined
```

try…except…

Python处理异常程序最简单的形式就是try…except…，类似于单分支选择结构。使用语法为：

```
try:
    <语句块1>    #可能会发生异常的代码，先执行以下试试看
except <异常类型>:
    <语句块2>    #如果try中代码抛出异常并被except捕获，执行此处代码
```

```
try:
    n = eval(input("请输入一个数字："))
    print(n)
except:
    print("输入错误请输入一个数字!")
```

try…except…else…

该结构可以看为双分支结构，else中的代码只有在try中代码没有引发异常的情况下执行。使用语法为：

try:

<代码块1> #可能会发生异常的代码

except <异常类型>:

<代码块2> #如果try中代码抛出异常并被except捕获，执行此处代码

else:

<代码块3> #try代码正常执行，则执行此处代码

```
try:
    n = eval(input("请输入一个数字："))
    print(n)
except:
    print("输入错误请输入一个数字!")
else:
    print("程序没有发生异常。")
```

try…except…finally…

该结构中，无论try代码中是否发生异常，finally语句块中的代码总会执行。使用语法为：

```
try:
    <代码块1>  #可能会发生异常的代码
except <异常类型>:
    <代码块2>  #处理异常代码
finally:
    <代码块3>  #总是执行
```

```
try:
    n = eval(input("请输入一个数字："))
    print(n)
except:
    print("输入错误请输入一个数字!")
finally:
    print("无论是否发生异常都会执行！")
```

捕捉多种异常的异常处理结构

实际开发中，同一段代码可能会抛出多种异常，并且针对不同异常进行相应的处理。使用语法为：

```
try:
    #可能会发生异常的代码
except <异常类型1>:
    #处理异常类型1的代码
except <异常类型2>:
    #处理异常类型2的代码
```

```
try:
    a=1
    n = eval(input("请输入一个数字"))
    print(a+n)
except NameError:
    print("变量未定义异常！")
except ZeroDivisionError:
    print("除零错误！")
except:
    print("其他异常！")
```