第 3 章 程序流程控制

Python 的结构化程序可分为顺序结构程序、分支结构程序和循环结构程序三种基本程序结构。

3.1 结构化程序设计

结构化程序设计可将复杂程序系统的设计转换为多个简单的独立模块的设计，是软件发展的一个重要的里程碑。

3.1.1 结构化程序设计方法

采用结构化程序设计方法构造的程序结构清晰，易于阅读、测试、排错和修改。由于每个模块执行单一功能，模块间联系较少，使程序设计更为简单，程序更可靠，并且每个模块可以独立设计和测试，进而增加了可维护性。

结构化程序设计是以模块功能和处理过程为主的详细设计来构造程序，任何程序都可由顺序、分支、循环三种基本结构构造。详细描述处理过程常用的三种工具主要有图

形、表格和语言。图形主要包括程序流程图、N-S 图和 PAD 图；表格主要包括判定表等；语言主要包括过程设计语言(PDL)等。

3.1.2 结构化程序设计的实施要素

在结构化程序设计的具体实施中，需要注意如下要素。

(1) 使用程序设计语言中的顺序、分支、循环等有限的控制结构表示程序的控制逻辑。

(2) 选用的控制结构只准有一个入口和一个出口。

(3) 程序语句组成容易识别的块，每个块只有一个入口和一个出口。

(4) 复杂结构应该用嵌套的基本控制结构进行组合嵌套实现。

(5) 对于语言中没有的控制结构，应该采用前后一致的方法模拟。

(6) 严格控制无条件转移 goto 语句的使用，经常可在下述情况下使用 goto 语句。

- 用一个非结构化的程序设计语言实现一个结构化的构造。
- 如果不使用 goto 语句将使得功能模糊。
- 在可以改善而不是损害程序可读性的情况下。

3.1.3 结构化程序的基本结构

结构化程序主要由以下三种基本结构组成。

1. 顺序结构

顺序结构是一种线性、有序的结构，它依次执行各语句模块。顺序型由几个连续的处理步骤依次排列构成，如图 3-1 所示。

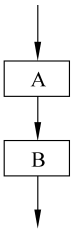


图 3-1 顺序型

2. 分支结构

虽然顺序结构程序能解决计算、输出等问题，但不能完成做判断后再选择的问题。例如，仅用顺序结构程序将不能够求解最大数问题和排序问题。对于需要先做判断再选择的问题就要使用分支程序结构。分支程序结构的执行是依据一定的条件选择执行路径，而不是完全按照语句出现的物理顺序执行。分支程序设计的关键是构造合适的分支条件和分析程序流程，根据不同的程序流程选择适当的分支语句。分支结构适合于带有逻辑或关系比较等条件判断的计算，例如若 x=0 成立，则选择路径 A，否则选择路径 B。通常，设计这类程序时需要先绘制其程序流程图，然后根据程序流程书写出源程序，这样可以将程序设计分析与程序语言分开，进而使得问题简单化和高效，更易于理解。

分支结构表明处理步骤出现了分支，需要根据某一特定条件选择其中的一个分支执行。分支结构主要分为单分支结构、双分支结构和多分支结构三种主要结构。

(1) 单分支结构。单分支结构是通过条件测试，当测试条件不成立时，则越过语句往下执行其他语句或结束，通常用于指定某一语句块是否执行。单分支结构的执行过程如图 3-2 所示。

(2) 双分支结构。双分支结构程序的执行过程是：当判断条件为真时，执行语句 1；

当判断条件为假时，执行语句 2，如图 3-3 所示。

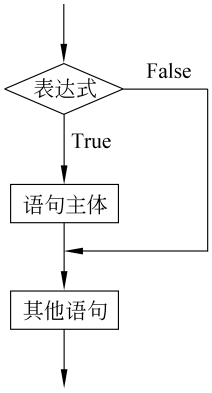


图 3-2　单分支结构的执行过程

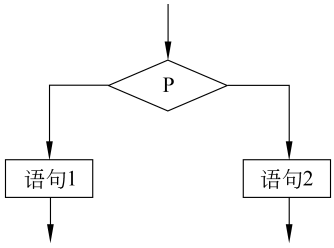


图 3-3　双分支结构

（3）多分支结构。多分支结构是双分支结构的扩展，通常设有 N 个条件和 N＋1 个语句块（或语句），测试条件从上向下测试到某个值为真时，执行对应的语句块，然后退出多分支结构去执行其他语句，如图 3-4 所示。

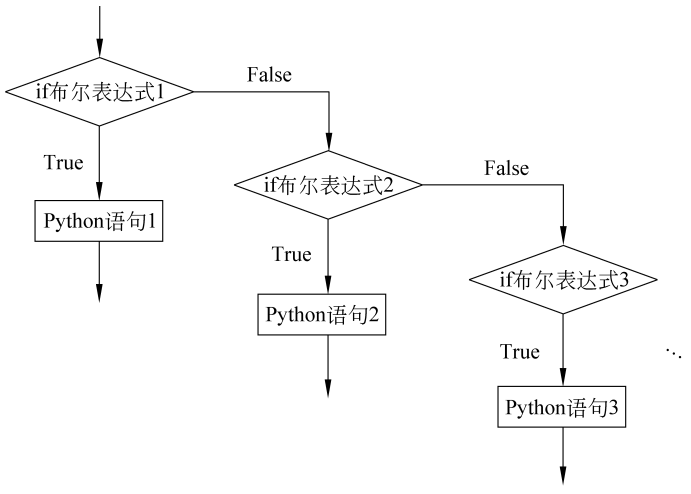


图 3-4　多分支结构

3. 循环结构

循环结构是指重复执行一个或几个模块，直到满足某一条件为止。常用的循环结构有 while 型循环结构和 until 型循环结构等。

（1）while 型循环结构。while 型循环结构是先判定循环条件，在循环控制条件成立时，再重复执行后续的特定处理，其执行过程如图 3-5 所示。

（2）until 型循环结构。until 型循环结构是后判定循环条件，在循环控制条件成立时，重复执行某些特定的处理，直到控制条件成立为止，其结构如图 3-6 所示。

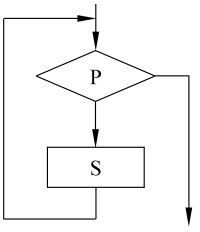


图 3-5　while 型循环结构

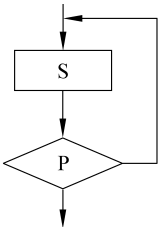


图 3-6　until 型循环结构

3.2　顺 序 程 序

顺序程序结构是指无分支、无循环的程序结构，在这种程序结构中，按语句的物理位置顺序执行程序。

3.2.1　简单语句

简单语句是指一种无分支、无循环的语句，顺序程序是由简单语句构成的。

1. 赋值语句

Python 的赋值语句是将一个变量绑定到某个对象来完成对变量的赋值。赋值语句有多种形式，主要包括一般形式、增量赋值形式、链式赋值形式和多重赋值形式等。Python 赋值语句的语法格式如下：

```
变量名=表达式
```

对于复杂的表达式，首先计算表达式的值，然后将变量指向计算结果。例如：

```
x=32+28                          #将变量 x 指向 32+28 的计算结果 60
s='hello'                        #将变量 s 指向 hello 字符串
[s,h]=['hello','Python']         #将列表元素 s 和 h 分别指向字符串 hello 和 Python
```

(1) 序列赋值。

在 Python 中，多个变量可以同时赋值，而不需要对一个变量赋值之后，再对另一个变量赋值，也就是说，多个变量可以同时赋值，使用一条语句就可以完成这个任务。

例如，序列赋值运算：

```
a,b,c=1,2,3
a,b=b,a
print(a,b,c)
```

程序运行结果如下：

```
2 1 3
```

可以看出，a 和 b 的值已完成交换，所以利用序列赋值可以进行两个或多个变量的值

交换。在 Python 中，交换工作称为序列解包或可选迭代解包，即将多个值的序列解开，然后将其存入变量序列中。序列解包允许函数返回一个以上的值并打包成元组，可以通过下面的例子进一步理解。

例如，序列解包：

```
nums=1,2,3
print(nums)
a,b,c=nums
print(a)
print(a,b,c)
```

程序运行结果如下：

```
(1,2,3)
1
1 2 3
```

在上述程序中，nums=1,2,3 完成元组打包，a,b,c=nums 完成序列解包，由输出结果可以看出，序列 nums 解包之后，变量 a、b、c 获得了对应的值 1、2、3。

应说明的是，解包序列中的元素量必须和放置在赋值符号“=”左边的变量数量完全一致，否则，在赋值时引发出错提示。

例如，在程序中，如果书写“a,b,c=1,2”，则出错提示为：

```
Traceback(most recent call last):
File"<pyshell#45>",line1in<module>
a,b,c=1,2
Value Error:not enough values to unparck(expected3,got2)
```

如果书写“a,b,c=1,2,3,4,5”，则出错提示为：

```
Traceback(most recent call last):
File"<pyshell#45>",line1,in<module>
>>>a,b,c=1,2,3,4,5
ValueError:too many values to unparck(expected3)
```

可以看出，由于左边元素的数量与右边元素的数量不相等，造成了执行结果的错误。其原因是左边变量个数多于右边元素个数(a,b,c=1,2)，没有足够的值解包；或者左边变量个数少于右边元素个数(a,b,c=1,2,3,4,5)，造成多个值没有解包。

可以给嵌套序列赋值：

```
>>>string='SPAK'
>>>(a,b),c=string[:2],string[2:]
>>>a,b,c
('S','P','AK')
```

赋值语句也可以将一系列整数赋给一组变量：

```
>>>red,green,blue=range(3)   #range(3)值为 0,1,2
>>>red,blue
(0,2)
```

(2) 链式赋值。

序列解包适用于对不同变量赋予不同值的情况,对于给不同变量赋予相同值的情况,可以采用链式赋值,即通过多个等式对多个变量赋予同一个值。

例如,对变量 a、b、c 进行多目标链式赋值。

```
a=b=c=20
print(a)
print(b)
print(c)
```

程序运行结果如下:

```
20
20
20
```

(3) 增强赋值。

增强赋值是指将表达式放在赋值运算符“=”的左边,例如将 x=x+1 写成 x+=1,这种写法对 *(乘)、/(除)、%(取模)等标准运算都适用。例如:

```
>>>a=6
>>>a+=2
>>>a
8
>>>a-=3
>>>a
5
>>>a*=2
>>>a
10
>>>a/=2
>>>a
5.0
```

可以看出,增强赋值是通过运算符来扩充完成,使赋值操作更为简单而有效。增强赋值的方法也适用于二元运算符的数据类型,例如:

```
>>>f='Hello,'
>>>f+='Python'
>>>f
'Hello,Python'
>>>f*=2
```

```
>>>f
'Hello,PythonHello,Python'
```

(4) 增强赋值语句总结。

① 增强赋值语句类型与功能描述。

x+=y：x+y→x,两个数相加。

x-=y：x-y→x,两个数相减。

x|=y：x|y→x,按位或运算。

x*=y：x*y→x,两个数相乘。

x^=y：x^y→x,按位异或运算。

x/=y：x/y→x,x 除以 y。

x>>=y：x>>y→x,右移位运算。

x%=y：x%y→x,返回除法的余数。

x<<=y：x<<y→x,左移位运算。

x**=y：x**y→x,返回 x 的 y 次幂。

x//=y：x//y→x,取整除返回商的整数部分。

② 增强赋值语句的特点。

- 使用增强赋值语句,致使程序更为简捷,输入量减少。
- 左侧只需计算一次。在 x+=y 中,y 可以是复杂的对象表达式。在增强形式中,则只需计算一次。然而,在完整形式 x=x+y 中,x 出现两次,必须执行两次。可以看出,增强赋值语句通常执行得更快。
- 增强形式可以自动执行对象的原处修改,提高了运算速度。

例如,列表的增强赋值：

```
>>>L=[1,2,3]
>>>L+=[4,5,6]
>>>L
[1,2,3,4,5,6]
```

由于 i+=1 的效率往往要比 i=i+1 更高,所以经常使用增强型赋值语句替换普通赋值语句,以此来优化代码。但并不是在任何情况下 i+=1 都等效于 i=i+1。

例如,使用增强型赋值语句：

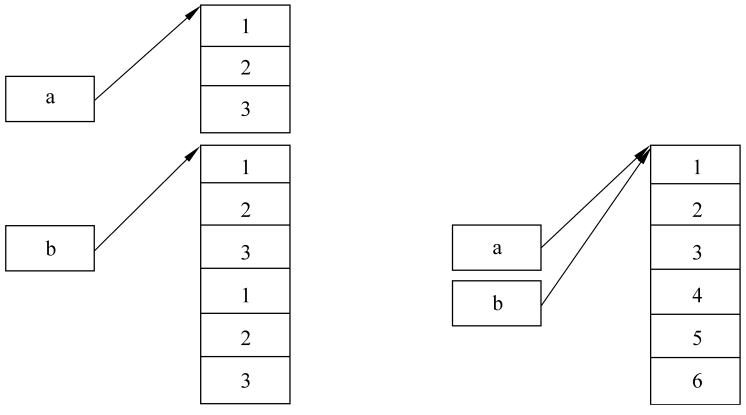
```
>>>a=[1,2,3]
>>>b=a
>>>b+=[1,2,3]
>>>print(a,b)
[1,2,3,1,2,3][1,2,3,1,2,3]
>>>id(a)                          #id(a)是对象 a 的存储地址
140213762276096
 >>> id(b)
 140213762276096
```

例如，使用普通赋值语句：

```
>>>a=[1,2,3]
>>>b=a
>>>b=b+[1,2,3]
>>>print(a,b)
[1,2,3][1,2,3,1,2,3]
>>>id(a)
140213762466232
>>> id(b)
140213762276168
```

上述的例子中，将一个列表类型对象赋值给变量 a，再将变量 a 赋值给变量 b，此时 a、b 指向了同一个内存对象[1,2,3]。然后分别应用增强赋值运算符和普通赋值运算符来操作变量 b。从最后的结果来看，使用增强型赋值语句的 a、b 在进行运算后依旧指向了同一个内存对象。但使用普通赋值语句则相反，a、b 分别指向了不同的内存对象，也就是说，隐式新建了一个内存对象。

使用增强赋值运算符操作可变对象（如列表）时可能会产生不可预测的结果。在 Python 中，允许若干个不同的变量引用指向同一个内存对象。增强赋值语句比普通赋值语句的效率更高，这是因为在 Python 源码中，增强赋值比普通赋值多实现了写回的功能，也就是说增强赋值在条件符合的情况下将以追加的方式来进行处理，而普通赋值则以新建的方式进行处理。这一特点导致了增强赋值语句中的变量对象始终只有一个，Python 解析器解析该语句时不会额外创建出新的内存对象，所以变量 a、b 的引用在最后依旧指向了同一个内存对象。相反，对于普通赋值运算语句，Python 解析器无法分辨语句中的两个同名变量（例如：b＝b＋1）是否应该为同一内存对象，所以再创建出一个新的内存对象用来存放最后的运算结果，导致 a、b 从原来指向同一内存对象，到最后分别指向了两个不同的内存对象，如图 3-7 所示。



(a) 普通赋值运算符　　(b) 增强赋值运算符

图 3-7　增强赋值运算符与普通赋值运算符的比较

2. pass 语句

在 Python 程序中，当要求语句不执行任何操作时，可以使用 pass 语句。pass 语句是一个空(null)操作语句，pass 语句不做任何事情，一般作为占位符来使用或者用于创建占位程序。

3. del 语句

del 语句的功能是解除变量和特性的绑定，并且移除数据结构(映射或序列)中的某部分，但不能用于直接删除数值，这只能通过垃圾收集来进行。

```
a=1                          #对象 1 被变量 a 引用
b=a                          #对象 1 被变量 b 引用
c=a                          #对象 1 被变量 c 引用
del a                        #删除变量 a,解除 a 对对象 1 的引用
del b                        #删除变量 b,解除 b 对对象 1 的引用
print(c)                     #最终变量 c 仍然引用对象 1
```

可以看出，使用 del 语句删除的是变量，而不是数据。

例如，对于列表 li=[1,2,3,4,5]，列表本身不包含数据 1、2、3、4、5，而是包含变量 li[0]、li[1]、li[2]、li[3]和 li[4]。first=li[0]是创建新的变量引用，而不是复制数据对象。

3.2.2　顺序程序设计

在 Python 程序中，语句执行的基本顺序是按各语句出现位置的先后顺序(物理顺序)执行，这种程序结构称为顺序程序结构，程序的运行轨迹是一条直线，无分支和循环出现。

如果某段程序由下述三条语句组成：

```
语句 1
语句 2
语句 3
```

如图 3-8 所示，程序执行顺序是先执行语句 1，再执行语句 2，最后执行语句 3，三个语句之间是顺序执行关系。

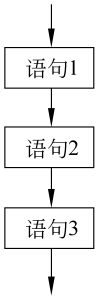


图 3-8　顺序程序结构

【例 3-1】 顺序程序结构。

```
#example3.1
a=10
b=20
c=30
d=a+b+c
print(a)
print(b)
print(c)
print(d)
```

程序运行结果如下：

```
10
20
30
60
```

3.3 分 支 程 序

基于分支结构的三种主要形式，可将 Python 的分支程序分为单分支程序、双分支程序和多分支程序。

3.3.1 单分支程序

单分支程序是按照单分支结构构造的程序。单分支 if 语句的语法格式如下：

```
if 条件
    语句块
后继语句
```

if 语句的功能是：首先完成条件判断，当测试条件成立时(非零)，则执行后面的语句块；否则跳过语句块，执行 if 语句的后继语句。其中条件是一个条件表达式，表达式后面是冒号“:”，表示一个语句块的开始，并且语句块做相应的缩进，一般是以 4 个空格为缩进单位。语句块是一条或多条语句序列。

在 Python 程序中，由于分支程序结构的执行是依据一定的条件测试来选择执行路径，所以必须掌握条件的设置方法。通过检测某个条件，达到分支选择。条件是一个表达式，测试的结果值为布尔型数据，即 True 或 False。

(1) 假(False)。False 是表示假的值，例如，FALSE、None、0、"(没有空格)、""(没有空格)、()、[]、{}都是假的值。

(2) 真(True)。除了上述假的值之外，其他的值都可以判定为真。可以使用命令行的运行方式，测试说明其真假值。

分支程序中的条件在多数情况下是一个关系比较运算。if 语句的判断条件可以用>(大于)、<(小于)、==(等于)、>=(大于或等于)、<=(小于或等于)等表示其关系。

例如，输入两个数字，找出其中最大的数(包括相等)输出。

```
x=eval(input("x="))
y=eval(input("y="))
if x>y:
    y=x
print("max:",y)
```

程序运行结果如下：

```
x=15
y=9
max:15
```

又如，使用列表作为条件表达式。

```
a=[1,2,3]
if a:                                    #使用列表作为条件表达式
    print(a)
```

程序运行结果如下：

```
[1,2,3]
```

3.3.2　双分支程序

双分支程序是使用较多的一种分支结构，其基本 if 语句的语法结构如下：

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

双分支 if 语句是在单分支 if 语句的基础上添加一个 else 语句，其含义是，如果 if 判断是 False，就不执行 if 语句块 1，而是执行语句块 2。else 之所以叫子句，是因为它不是独立的语句，而只能作为 if 语句的一部分，当条件不满足时执行它。

例如，如果输入数为 10，则输出 true；否则输出 false。

```
x= eval (input('Input a numbers: '))
if x==10:
    print("true")
else:
    print("false")
```

【例 3-2】 输入两个不相等的数字，处理后输出其中较大的数字。

```
#example3.2
x=eval(input('输入第 1 个数字:'))
y=eval(input('输入第 2 个数字:'))
print('输入的两个数字: ',x,y)
if x>y:
    print('较大数字:',x)
else:
    print('较大数字:',y)
```

程序运行结果如下：

```
输入第 1 个数字:22
输入第 2 个数字:55
输入的两个数字:22 55
较大数字:55
```

3.3.3 多分支结构

当判断的条件有多个且判断结果有多个的时候,可以用多分支 if 语句进行判断,其结构如图 3-3 所示,多分支 if 语句的语法格式如下:

```
if 条件 1
    语句块 1
elif 条件 2
    语句块 2
elif 条件 3
    语句块 3
…
elif 条件 n
    语句块 n
else
```

在上述格式中,使用了 elif 子句。elif 是“elseif”的简写,表示 if 和 else 子句的联合使用,它是具有条件的 else 子句。if 语句执行是从上向下判断,如果某个判断结果是 True,则执行该判断所对应的语句块,当然也就忽略掉剩下的 elif 和 else。

【例 3-3】 判断年龄范围程序。

```
#example3.3
Age=int(input('age='))
if age>18:
    print('>18')
elif age>6:
    print('18>=age>6')
else:
    print('<=6')
```

程序运行结果如下:

```
age=18
18>=age>6
======================
age=6
<=6
======================
age=15
18>=age>6
```

```
======================
age=-1
<=6
```

if 判断条件还可以简写如下：

```
if x:
    print('True')
```

只要 x 是非零数值、非空字符串、非空 list 等，就判断为 True，否则为 False。

3.3.4　分支结构的嵌套

当 if 语句主体中又包含 if 语句时，就称这个语句为嵌套 if 语句，或称为分支结构的嵌套。在程序结构上，嵌套 if 语句就是将 if…elif…else 结构放在另外一个 if…elif…else 结构中，其一般格式如下：

```
if 表达式 1:
    if 表达式 2:
        语句块
    elif 表达式 3:
        语句块
    else:
        语句块
    elif 表达式 4:
        语句块
else:
    语句块
```

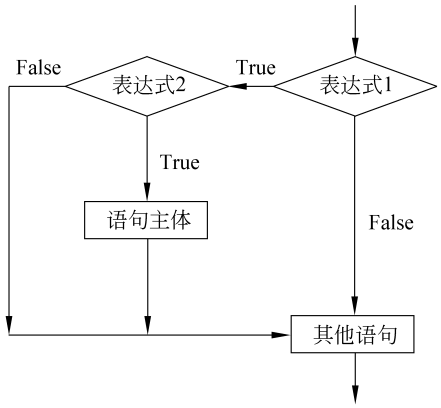


图 3-9　嵌套 if 语句的基本流程

嵌套 if 语句可以用流程图 3-9 表示。

【例 3-4】　在 if 语句中嵌套 if 语句。

```
#example 3.4
x=int(input('Please enter an integer in 0-10:'))
if x>=0:
    if x>9:
        print('x>9')
    elif x==0:
        print('x=0')
    else:
        print('9>x>0')
else:
     print('x<0')
```

程序运行结果如下：

```
Please enter an integer in 0-10:10
x>9
```

```
======================
Please enter an integer in 0-10:-1
x<0
>>>
======================
Please enter an integer in 0-10:0
x=0
>>>
======================
Please enter an integer in 0-10:1
9>x>0
>>>
======================
Please enter an integer in 0-10:7
9>x>0
======================
```

3.4 循环程序结构

循环程序是指在给定的条件为真的情况下，重复执行某些语句。它是程序设计中的一种重要结构。应用循环结构可以减少程序中大量重复的语句。Python 语言的循环结构主要包含两种类型，分别是由 while 语句实现的 while 循环和由 for 语句实现 for 循环。这两种循环语句是编程的基本元素，例如，当需要用户输入十个整数时，如果使用顺序结构，则需要使用十条输入语句，但是使用循环结构，只需要一条输入语句就足够了。由此可见，循环结构能够给程序设计开发带来极大的便利与高效，使设计的程序更为简洁。

Python 语言中涉及循环程序设计的常用语句主要有：while 语句、for 语句以及与 for 语句一起使用的 range()内置函数。与此同时，还包括与循环语句紧密相关的 break 语句、continue 语句和 pass 语句等。

3.4.1 while 循环程序

while 循环程序主要由 while 语句构成，while 语句的功能是：当给定的条件表达式为真时，重复执行循环体(即内嵌的语句块)，直到条件为假时才退出循环，并执行循环体后面的语句。while 语句的语法格式如下：

while 条件表达式：

循环体

while 语句的工作流程图如图 3-10 所示。

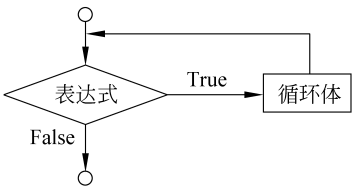


图 3-10 while 语句的流程图

将 while 语句的流程图与 if 语句的流程图相比较后可以看出，两者都由一个表达式和语句体或循环体组成，并且都是在表达式的值为真时执行语句体或循环体。但两者的关键区别是，对于 if 语句，它执行完循环体后，就退出了 if 语句；而对于 while 语句，它执行完循环体后，又返回表达式，只要表达式的值为真，它将一直周而复始地重复这一过程。

关于 while 语句的几点说明如下。

- 保持组成循环体的各语句的缩进格式。
- 循环体中要有控制循环结束的代码，否则会造成无限循环。
- 循环体既可以由单条语句组成，也可以由语句块组成，但是不能没有任何语句。
- 因为 Python 语言区分大小写，关键字 while 必须为英文小写。

例如，在下面的列表赋值中，将 while 循环中的序列分割为开头和剩余的两部分。

```
>>>L=[1,2,3,4]
>>>while L:                    #当 L 为空时，结束循环
       front,L=L[0],L[1:]
       print(front,L)          #单击两次 Return 键
1 [2,3,4]
2 [3,4]
3 [4]
4 []
```

【例 3-5】 计算并输出 1～20 之间的奇数的程序。

```
#example3.5
integer=1
while integer<=20:
    if integer%2==1:
        print(integer)
    integer=integer+1
```

程序运行结果如下：

```
1
3
5
7
9
11
13
15
17
19
```

【例 3-6】 打印斐波那契数列的前 n 个元素的程序。

```
a=0
b=1
```

```
sum=0
n=int(input('n='))
while n>0:
    sum=a+b
    a=b
    b=sum
    n-=1
        print(a,end='  ')
```

程序运行结果如下：

```
n=10
1 1 2 3 5 8 13 21 34 55
```

当使用循环结构时，需要考虑控制循环结束的方法。对于 while 语句，通常使用下述两种方式来控制循环的结束：一种是计数器循环控制法，一种是信号值循环控制法。

1. 计数器循环控制法

计数器控制的循环结构适于在循环执行之前就需要知道重复执行次数。例如，要求用户输入 10 个整数，每次输入一个数字之后，求出其平均值并输出结果。使用计数器来控制输入循环必须设置一个变量 counter 作为计数器，可以用它来控制输入语句的执行次数。计数器一旦超过 10，便停止循环。此外，还需要一个变量 total 来累计输入整数的次数，将变量 total 初始化为 0。

程序运行过程如下：首先，用户输入 10 个整数。用一条 while 语句使函数循环执行 10 次。循环语句中的表达式为：counter＜＝10，因为 counter 的初始值为 1，而循环体中使循环趋向于结束的语句是：counter＝counter＋1，所以循环体将执行 10 次。

每轮循环中，函数会输出“输入一个整数：”，提示用户进行输入。当用户输入后，int()函数将输入的内容转换为一个整数，并累加到变量 total 中。这三个动作是用一条语句完成的。

【例 3-7】 计算输入数据平均值的程序。

```
#example3.7
total=0
counter=1
while counter<=10:
    total= total+int(input('input a int data:'))
    counter=counter+1
print('average value:',float(total)/10)
```

首先将累加的结果转换为浮点数，然后除以 10，并用 print()函数输出。如果使用计数器 counter 除以累加值 total 计算平均值，将导致错误。因为当用户输入第十个整数时，counter 的值为 10，表达式值为真，所以循环体继续执行。当执行了循环体的最后一条语句即 counter＝counter＋1 之后，counter 的值变成 11，再次判断表达式，这时表达式的值为假，所以退出循环。也就是说，当循环退出时，counter 的值是 11，而不是 10。所

以,用它来求 10 个整数的平均值显然是错误的。

程序运行结果如下:

```
input a int data:2
input a int data:3
input a int data:3
input a int data:3
input a int data:3
input a int data:3
input a int data:3
input a int data:3
input a int data:3
input a int data:5
average value:3.1
```

2. 信号值循环控制法

计数器循环控制法适合于事先能确定循环次数的场景,但是当无法事先确定具体的循环次数时,就需要使用信号值循环控制法。例如,设计一段程序来计算某计算机学院的各系教师的平均年龄。可以使用一个循环语句来录入各人员的年龄,但是由于各系人员数不一致,计数器循环控制法不适合这种场景,这时可以使用信号值循环控制方法。信号值就是使用一个特殊数值,用它来控制循环结束。

在使用信号值循环控制法的程序中,可以不断地输入各系人员的年龄,直到输入结束时就可以输入信号值,告诉程序输入各系人员年龄的工作结束了。因为信号值跟正常的数据一起输入,所以选择信号值时一定要使信号值与正常的数据有明显的区别,以防止与正常的值相混淆。例如,各系人员的年龄都大于或等于 18 岁,为了防止与正常的值相混淆,选择 1 作为信号值,这样就绝对不会产生混淆。

【例 3-8】 使用信号值循环控制的平均值计算程序。

```
#example3.8
total=0                          #用变量 total 存储年龄之和
counter=0                        #用 counter 存储人员数量
age=int(input('输入人员年龄,用 1 表示输入结束:'))
while age!=1:
    total=total+age
    counter=counter+1
    age=int(input('输入人员年龄,用 1 表示输入结束:'))

if counter!=0:
    print('平均年龄是:',float(total)/counter)
else:
    print('输入完成')
```

程序运行结果如下:

```
输入人员年龄,用 1 表示输入结束:20
输入人员年龄,用 1 表示输入结束:40
输入人员年龄,用 1 表示输入结束:60
输入人员年龄,用 1 表示输入结束:1
平均年龄是: 40.0
```

如果 counter 变量的值为 0,那么执行上述选择结构中 else 子句的内嵌语句,即输出“输入完成!”。

在循环体中,将输入的年龄累加到变量 total 中,并将计数器加 1。接着执行循环体中的最后一条语句:要求用户再次输入一个人的年龄。需要注意的是,对 while 结构的条件进行判断之前先请求下一个值,这样就能先判断刚才输入的值是否是信号值,再对该值进行处理。当循环体中的语句执行一遍后,程序会重新检测 while 语句的条件表达式,以决定是否再次执行 while 结构的循环体。换句话说,如果刚才输入的值是信号值,则退出循环体;否则,继续重复执行循环体。只要循环体执行一次,那么当退出循环后,统计人员数量的变量 counter 的值肯定大于 0,所以这时就会执行最后面的选择结构中的 if 子句内嵌的语句体,即计算平均年龄并输出。

3.4.2 for 循环

for 循环是一种遍历型的循环,因为它依次对某个序列中全体元素进行遍历,遍历完所有元素之后便终止循环。for 语句的语法格式如下:

```
for 控制变量 in 可遍历的表达式:
循环体{
```

其中,关键字 in 是 for 语句的组成部分,为了遍历可遍历的表达式,每次循环时,都将控制变量设置为可遍历的表达式的当前元素,然后在循环体开始处执行。当可遍历的表达式中的元素遍历一遍之后,即没有元素可供遍历时,就退出循环。for 语句的工作流程图如图 3-11 所示。

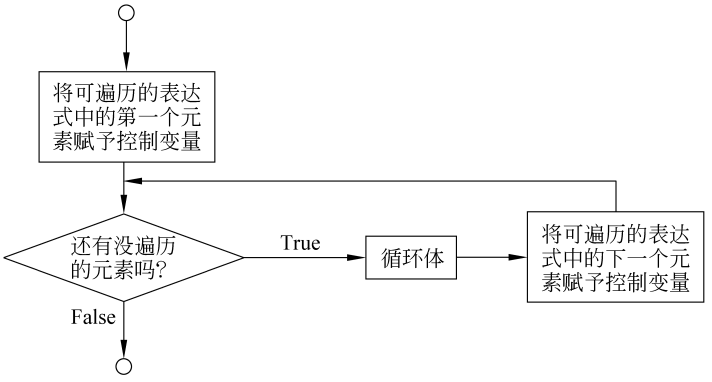


图 3-11 for 语句的工作流程图

例如：

```
for char in 'hello':
    print (char)
```

程序运行结果如下：

```
h
e
l
l
o
```

1. for i in range()结构

可将 for 语句与 range()函数结合使用，构成 for i in range()结构。例如，用于输出 0～9 之间的偶数的程序如下：

```
#输出 10 以下的非负整数中的偶数
for integer in range(10):
    if integer % 2==0:
    print(integer)
```

程序运行结果如下：

```
0
2
4
6
8
```

上述程序的执行过程说明如下：首先，for 语句开始执行时，range()函数会生成一个由 0～9 这十个值组成的数字序列。然后，将序列中的第一个值即 0 赋给变量 integer，并执行循环体。在循环体中，将变量 integer 除以 2，如果余数为零，则打印该值；否则跳过打印语句。执行循环体中的选择语句后，将序列中的下一个值装入变量 integer，如果该值是序列中的，那么继续循环，以此类推，直到遍历完序列中的所有元素为止。

【例 3-9】 打印九九乘法表的程序。

```
#输出九九乘法表
    for i in range(1, 10):
    for j in range(1, i+1):
        print('{}x{}={}\t '.format(j, i, i * j), end='')
    print('')
```

程序运行结果如下：

```
1×1
1×2=2  2×2=4
```

```
1×3=3  2×3=6   3×3=9
1×4=4  2×4=8   3×4=12  4×4=16
1×5=5  2×5=10  3×5=15  4×5=20  5×5=25
1×6=6  2×6=12  3×6=18  4×6=24  5×6=30  6×6=36
1×7=7  2×7=14  3×7=21  4×7=28  5×7=35  6×7=42  7×7=49
1×8=8  2×8=16  3×8=24  4×8=32  5×8=40  6×8=48  7×8=56  8×8=64
1×9=9  2×9=18  3×9=27  4×9=36  5×9=45  6×9=54  7×9=63  8×9=72  9×9=81
```

2. for e in L 结构

在 for e in L 结构中，L 为一个列表。与上述的 for i in range()结构不同的是，如果循环中 L 被改变了，将会影响到 for e in L 结构。

例如，如果需要遍历列表 L，并打印出 L 中的所有元素，还要在元素为 0 时向列表中添加元素 100，使用 for e in L 结构的方法如下。

(1) 直接使用 append()函数在原列表中添加新元素 100，程序如下：

```
L=[0,1,2,3,4,5]
for e in L:
    print(e,end=' ')
    if e==0:
        L.append(100)
```

程序运行结果如下：

```
0 1 2 3 4 5 100
```

(2) 可以使用 L=L+[100]这种方式添加新元素 100，程序如下：

```
L=[0,1,2,3,4,5]
for e in L:
    print(e,end=' ')
    if e==0:
    L+= [100]
```

程序运行结果如下：

```
0 1 2 3 4 5 100
```

3.4.3 跳出循环

使用 break 语句和 continue 语句可以改变循环流程。当在循环结构中执行 break 语句时，将导致立即跳出循环结构，转而执行该结构后面的语句。使用 break 语句可以打破最小封闭的 for 或 while 循环。

可以使用 break 语句终止循环语句，即在循环条件没有 False 条件或者序列还没被完全循环结束的情况下，也可停止执行循环语句。

1. break 语句

在 while 和 for 循环中，如果使用嵌套循环，break 语句将停止执行最深层的循环，并