第 3 章

流 程 控 制

Python 语言中如何表示逻辑条件，根据逻辑条件选择做什么或者不做什么？用什么样的语句来实现选择结构？在逻辑条件下通过重复执行语句或语句块，直到逻辑条件不再满足。又需要学习哪些语句才能帮助我们解决在工作或生活中遇到的复杂问题。这些问题是本章我们要重点解决和学习的问题。

本章主要内容：

- if 语句和分支结构，让判断更加智能；
- for 和 while 循环结构，实现更多重复操作、更好的解决方法；
- 迭代和列表解析，实现更优雅地简化代码。

3.1 if 语 句

3.1.1 问题的提出

“凡事预则立，不预则废。”每天我们先做什么，后做什么，都可以计划好、安排好。但生活中也会发生一些无法预料的小概率事件，经常打乱我们学习或工作的节奏，我们需要在这些繁多的事务中优先选择需要处理的紧急而重要的事务，而把一般的事务放在后面处理，这就需要我们做出正确的选择。对于人类而言，可以经过后天的学习和实践做出正确的判断与选择；对于计算机的世界，怎么让计算机像人类一样，具备一定的判断能力和处理能力呢？接下来，我们将学习如何使用 if 语句来让计算机像人类一样做出简单或复杂的思考和判断。

if 语句，又称为条件语句，属于程序控制结构中的选择结构（又称为条件结构）。if 语句根据条件表达式的逻辑结果做出选择，决定执行什么语句。

3.1.2 if 语句基本结构

1. 简单分支语句

简单分支语句又叫单分支选择结构，适用于处理相对简单的事务，比如解决条件满足(达到)的情况下做什么的问题，例如：用户输入了错误的密码，用户界面会给出错误的提示信息；学生成绩未达到及格线，会提示学生该门课程不及格。

简单分支语句的语法形式如下：

```
if  条件表达式:
    语句块
```

简单分支语句由关键词 if 开头，后面是条件表达式，条件表达式后面是一个英文字符冒号“:”，初学者在书写或编辑代码时，容易遗漏这个重要的符号。

语句块是满足条件或达到条件时要执行的代码。可以是简单的一条语句，也可以是复杂的语句，比如可以嵌套其他控制结构的语句用来实现更为复杂问题的处理。

为了明确语句块是条件表达式成立或满足时才能够执行的代码，语句块必须遵循 Python 语法规则进行缩进处理，一般是 4 个空格字符的宽度(详见第 2 章 2.1.1 小节用缩进表示代码块)。

在代码运行到简单分支语句时，Python 解释器会根据条件表达式的运算结果进行判断：如果条件表达式的值为 True(真/成立)，那么执行语句块。如果条件表达式的值为 False(假/不成立)，那么语句块不会被执行，如图 3.1 所示。

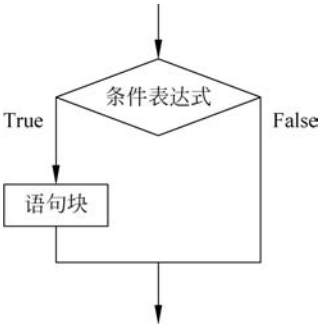

图 3.1　简单分支语句流程图

条件表达式可以是简单的关系表达式，也可以是复杂的关系表达式，还可以是一个具体的逻辑值(True 或 False)。

```
if  True:
    print ("根据条件表达式的值,你会看到我")
```

上面的代码在实践中是没有意义的，因为条件永远为真(True)，输出语句都要被执行。但像用变量构成的关系表达式，比如“a>0”就有很大的不确定性，需要根据变量 a 的值与 0 进行关系运算得出的逻辑值来进行判断，这个交给计算机去判断和执行，它会做得更好、更有效率。这也是为什么说学习编写程序代码能够开发大脑的潜能，提高逻辑思维能力和解题能力的原因之一。请读者结合本章的案例，多练习、多上机操作实践；多思考，开发大脑的潜能，编写出解决问题的 Python 代码，从而提高解决问题的能力。

简单分支语句又称为单分支语句，是一种“如果……那么”的简单分支结构，例如：判断学生成绩是否及格，学生成绩在 60 分以上的给出“及格”的信息。

```
score = 60
if score > = 60:
  print("及格")
```

结果：及格

2. 双分支语句

双分支语句根据条件表达式的值提供了解决问题的两种思路，即条件表达式成立时做什么，不成立时做什么，如图 3.2 所示。

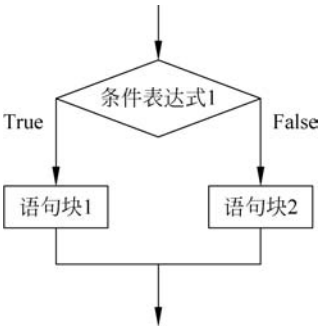

图 3.2　双分支语句流程图

双分支语句的语法形式如下：

```
if  条件表达式:
    语句块 1
else:
    语句块 2
```

下面我们使用双分支语句来判断学生成绩是否及格，大于等于 60 分的给出“及格”的信息，小于 60 分的给出“不及格”的信息。

```
score = 56
if score > = 60:
   print("及格")
else:
   print("不及格")
```

结果：不及格

3. 多个条件判断语句

多个条件判断语句的语法形式如下：

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

在多个条件判断语句中，当条件表达式 1 成立时，执行语句块 1 的代码，否则继续判断条件表达式 2，如果条件表达式 2 成立则执行语句块 2；当条件表达式 1 和条件表达式 2 都不成立时，执行语句块 3 的内容。使用 if 语句需要注意的是：每个条件表达式后面要使用冒号“:”，满足条件要执行的语句块以缩进的形式呈现。根据解决问题的需要，elif 和 else 可以灵活进行组合使用，也可以省略掉这两个部分的内容，构成最简化的单分支语句。

在 Python 语言中，elif 是 elseif 的缩略用法，表示否则如果的意思，也就是继续判断条件表达式 2 是否满足条件。“else：”后面的语句块 3 表示：当条件表达式 1 和条件表达式 2 都不成立时，执行语句块 3 的语句，如图 3.3 所示。

多个条件判断语句可以解决生活中遇到的复杂问题，例如：根据学生成绩给出简单的评价，70 分以上为“良”，大于等于 60 分并且在 70 分以下为“及格”。

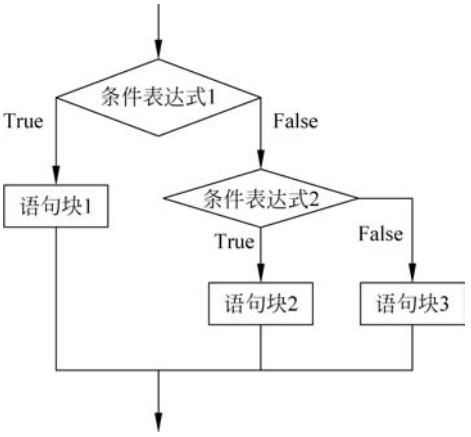

图 3.3　多个条件判断语句的流程图

```
score = float(input("输入学生的成绩: "))
if  score >= 70:
    print ("良")
elif  score >= 60:
    print ("及格")
else:
    print ("不及格")
```

请读者在运行程序时输入三组以上的测试数据：分数在 60 分以下、60～70 分、70 分以上。

4. 使用多个 elif 完善学生成绩评价程序

在实际工作中，我们遇到的问题更复杂，处理业务时需要分析和判断的逻辑条件更多。以学生成绩评价为例，不仅要判断学生成绩是否及格，还需要更多的评价机制才能客观地反映学生成绩的现状。我们需要对学生的成绩的评价程序进行完善和修改，使之更符合实际需要。

【例 3.1】 使用多个 elif 来完善对学生成绩的评价。

```
#接收键盘上输入的数据,并将输入的内容转换为浮点型对象
score = float(input('请输入学生成绩: '))
if  score >= 90  and score <= 100:
     print ('优秀')
elif score < 90  and score >= 80:
     print ('良好')
elif score < 80  and score >= 70:
     print ('中等')
elif score < 70  and score >= 60:
     print ('及格')
else:
    print ('不及格')
```

例题分析：在本例中，使用了多个 elif 语句进行条件判断，对于成绩的判断采取了从高分到低分进行分数区间的判断。这样做能够避免因为判断条件的不合理设计而得出错误结果的情况发生。

3.1.3　真值测试

在 if 或者 while 语句中，条件表达式的值可能为 True 或者 False。在编写流程控制的代码时，编程者应熟悉各种条件表达式的真值。

在 Python 语言中，对于非零的数字或非空的对象，真值为真，返回值为 True。

对于数值 0、0.0、0j、空的对象（包括空字符、[]、()、{}、None、False、set()、range(0)）其真值为假，返回值为 False。

Bool()函数可以返回对象的布尔值：True(真)或 False(假)。例如：

```
>>> bool([]),bool(()),bool({})                    #返回空的列表、元组和空字典的布尔值
```

结果：

```
(False, False, False)
>>> bool("")                                       #返回空字符串的布尔值
```

结果：

```
False
>>> bool(0)                                        #返回数值 0 的布尔值
```

结果：False

关于真值测试，表 3.1 和表 3.2 给出了常见的关系运算符和布尔运算符的真值测试示例，可供读者在学习和使用过程中参考。

表 3.1　关系运算符的真值测试

运 算 符	返回值	示　例	

==（等于）	True 或 False	>>> x = 3
		>>> y = 2
		>>> x == y
		False
			>>> ch1 = "A"
			>>> ch2 = "A"
			>>> ch1 == ch2
			True

>（大于）	True 或 False	>>> 3 > 2 True	>>> 2 > 5 False

<（小于）	True 或 False	>>> 5 < 10
		True
		>>> 5 < 3
		False
			>>> 'a'<'b'
			True
			>>> 'c'<'a'
			False

>=（大等于）	True 或 False	>>> 3 >= 0 True	>>> 3 >= 6 False

<=（小等于）	True 或 False	>>> 5 <= 6
		True
		>>> 5 <= 5
		True
			>>> 5 <= 3
			False

!=（不等于）	True 或 False	>>> 3!= 2 True	>>> 2!= 2 False

表 3.2 布尔运算符的真值测试

运 算 符	描 述	示 例	

and （“与”）	x and y 如果 x 和 y 都为真，就是真	>>> x = True
		>>> y = True
		>>> x and y
		True
			>>> x = 3 > 2
			>>> y = 3 > 5
			>>> x and y
			False

or （“或”）	x or y 如果 x 或 y 为真，就是真	>>> x = True
		>>> y = False
		>>> x or y
		True
			>>> x = False
			>>> y = True
			>>> x or y
			True

not （“非”）	not x 如果 x 为假，那就是真
	not x 如果 x 为真，那就是假
		>>> x = False
		>>> not x
		True
			>>> x = True
			>>> not x
			False

3.1.4 if else 三元表达式

Python 语言有自己特殊的三元表达式，在编程实践中可以采用 if else 来简化代码的书写，if else 三元表达式的语法为

```
条件为真时的结果 if 判断的条件 else 条件为假时的结果
```

语法说明：如果判断的条件逻辑值为真，输出 if 前面的内容，即“条件为真时的结果”。如果判断的条件逻辑值为假，输出 else 后面的内容，即“条件为假时的结果”。下面以具体案例进行说明。

【例 3.2】 已知两个数分别存放在变量 x 和 y 中，将两个数中的最大数存放在变量 z 中。

```
x,y = 3,5                                    # 使用元组的方法给变量 x 和 y 分别赋值为 3 和 5
if x > y:
    z = x
else:
    z = y
print (f"{x}和{y}之间的最大数是{z}")
```

结果：3 和 5 之间的最大数是 5

【例 3.3】 使用三元表达式对例 3.2 的代码进行优化。

```
# 已知两个数分别存放在变量 x 和 y,将两个数中的最大数存放在变量 z 中
x = 3
y = 5
z = x if (x > y) else y                      # if else 三元表达式
print (z)                                    # 输出最大数的值
```

结果：5

【例 3.4】　判断一个数是否为偶数。

```
x = 3
print (f"{x}是偶数"  if  (x % 2 == 0)   else  f"{x}不是偶数" )
```

结果：3 不是偶数

小结：使用 if　else 可以替代三元表达式，实现三元表达式的功能，能够简化代码的书写，在相对简单的条件判断中，可以让代码显得更简洁、更优雅，这也是 Python 语言编程的魅力。

3.2　for　语　句

for 循环语句又称为 for 循环结构，在 Python 语言中，是通过遍历的形式访问任何有序的序列对象内的元素，来实现重复执行指定次数的操作。比如遍历字符串、列表或元组、字典的键值等数据对象。对于循环次数固定的重复操作我们可以使用 for 语句来实现。for 语句更多地用于一些根据对象元素长度或需要遍历访问所有可迭代对象的场合。

3.2.1　for 循环基本格式

for 循环的基本语法为

```
for  <赋值目标/目标>  in  <可迭代对象>:
    <语句块>
```

语法说明：for 循环语句的首行定义了赋值的目标和遍历(访问)的可迭代对象，冒号后面以缩进形式表示要重复执行的语句块。可迭代对象包括迭代器、字符串、列表、元组、字典、集合等。for 循环的流程图如图 3.4 所示。

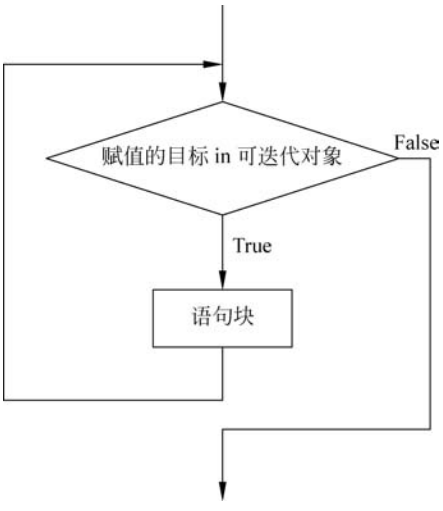

图 3.4　for 循环的流程图

在 Python 语言中，for 循环又称为迭代循环，当运行到 for 循环时，Python 解释器会逐个将可迭代对象中的元素赋值给目标，然后为每个元素执行循环语句块。循环语句块则使用赋值目标来引用可迭代对象中当前的元素，直到遍历完可迭代对象的所有元素。下面的案例给出了具体的使用方法。

【例 3.5】 for 循环输出指定字符串信息 3 次。

```
for i in range(3):
  print ("勤洗手、常通风、戴口罩")          #重复执行的语句
```

结果：

```
勤洗手、常通风、戴口罩
勤洗手、常通风、戴口罩
勤洗手、常通风、戴口罩
```

例题分析：range(3) 迭代器生成的可迭代的序列，由 0,1,2 组成，一共有三个元素，for 语句定义了赋值的目标变量 i，Python 解释器自动在每次循环前将可迭代序列的元素依次传递给赋值的目标变量，并进行成员运算，根据成员运算的结果决定是否执行循环语句块。例 3.5 中，在循环语句中虽然没有直接使用 i 的值，但在遍历可迭代序列 0,1,2 期间一共输出了三次字符串信息。

【例 3.6】 for 循环遍历字符串。

```
str1 = '123'
for ch in str1:
    print(ch)
```

结果：

```
1
2
3
```

例题分析：ch 作为赋值目标承担了遍历字符串“123”的重任，本例可以理解为 ch 的值依次为字符串的 s[i]值。i 的取值范围为(0,1,2)，具体分析如表 3.3 所示。

表 3.3 例 3.6 for 循环遍历字符串情况分析

循环次数	赋值目标 ch 的值	循环语句块 print(ch)
1	'1'	输出：'1'
2	'2'	输出：'2'
3	'3'	输出：'3'

【例 3.7】 for 循环遍历列表对象。

```
list1 = ['1','2','3']
for n in list1:
   print(n)
```

结果：

```
1
2
3
```

【例 3.8】 for 循环遍历列表的另外一种方法。

```
for i in range(len(list1)):
   print (list1[i])
```

结果：

```
1
2
3
```

例题分析：通过遍历 range() 函数生成的序列可以通过访问列表的索引号输出列表的各个元素。在案例中，使用 len()函数对列表 list1 元素总个数进行统计，统计的结果作为 range()函数的参数，生成序列 0，1，2 ，这组序列值刚好对应列表的索引号，因此可以用访问列表索引号的方式输出列表的各个元素。这种方法和用成员运算符进行判断的思路异曲同工，但具有更大的灵活性，比如按索引号输出特定要求的列表元素。

【例 3.9】 逆序输出列表中的元素。

```
list1 = ['1','2','3']
for i in range(len(list1) - 1, - 1, - 1):
  print(list1[i])
```

结果：

```
3
2
1
```

3.2.2　多个变量迭代

在编程实践中，为了使编写的代码更简洁、高效，并提高内存的使用效率、降低内存的占用，Python 语言提供了多个变量进行遍历（迭代）的方法。读者可以先熟悉具体用法，然后在后续的编程实践中学习使用。

【例 3.10】 对字典的键、值进行迭代，输出字典的键值信息。

```
word_dict = {"if": "条件","for": "迭代循环","while": "条件循环"}
for  word,explain   in  word_dict.items():
    print (word,"表示 ",explain)
```

结果：

```
if 表示　条件
for 表示　迭代循环
while 表示　条件循环
```

例题分析：字典对象的 items()方法返回的是字典的键值组合数据对象，又称为字典视

图对象，是可迭代的数据类型。通过使用多个变量迭代的方式，本例中以变量名 word 和 explain 作为迭代变量来遍历该对象，实现对 word_dict 字典对象的字典键、值信息的输出。

【例 3.11】 输出逻辑按位“与”运算的真值表。

```
p = [0,0,1,1]
q = [0,1,1,0]
for m,n in zip(p,q):
print (f"{m}& {m} = {m & n}")
```

结果：

```
0 & 0 = 0
0 & 0 = 0
1 & 1 = 1
1 & 1 = 0
```

例题分析：本例中使用 zip() 函数将可迭代对象 p 和 q 作为参数，在 Python 3 中为了减少内存，zip()返回一个可迭代的对象(由元组构成列表的特殊对象)，无法直接查看，但可进行迭代操作。可以使用下面的代码将 zip()返回的对象转换成列表数据进行查看。

```
>>> print(zip(p,q))                          # 直接输出 zip()函数返回的对象，显示对象的地址信息
< zip object at 0x000002216BE67A00 >
>>> list(zip(p,q))                           # 转换成列表对象，可查看数据内容
[(0, 0), (0, 1), (1, 1), (1, 0)]
```

读者可以根据例 3.11，修改代码实现输出逻辑“或”运算的真值表，以检验学习成效。

3.2.3 break 和 continue 语句

为了实现 for 循环过程中的灵活性、智能性，Python 语言吸收了其他高级语言的特点，引入了 break 和 continue 语句，结合 if 语句进行使用，用于中断循环或者跳过循环语句块，继续执行循环。

for 语句的完整语法格式如下：

```
for  <赋值目标/目标>  in  <对象>:
     <语句块 1 >
     If   <条件>: break
     if   <条件>:   continue
else:
     <语句块 2 >
```

【例 3.12】 使用 for 语句和 continue 语句实现列表元素的去重功能。

```
alst = [1,2,1,2,3,4,5,1]
blst = [ ]
for n in alst:
    if  n in blst:                          # 元素 n 在列表 blst 里面
        continue                            # 跳过循环语句块，继续执行循环
    blst.append(n)
print (blst)
```