第3章

程序控制结构

学习目标：

- 学会用传统流程图和 N-S 流程图表示问题求解的算法。
- 学习并掌握选择结构的流程，能熟练应用单分支、双分支、多分支和嵌套分支的 if 语句编写程序。
- 学习并掌握循环结构的流程，能熟练使用 while 和 for 语句编写程序。
- 学习并掌握循环的退出机制，能熟练应用 break 和 continue 语句。
- 学习并掌握循环的嵌套，能熟练应用多重循环解决较复杂的问题。

3.1 控制结构概述

3.1.1 处理模式

从实际中可发现，人类处理问题的方式具有某些固定的模式化特征。下面举个生活中的例子加以说明。

【例 3-1】 生活中的处理现象。

小明早上起床，洗漱完毕，吃些早点，然后准备去学校。走到门口，准备像往常一样穿上皮鞋，但想到今天有体育课，就穿了运动鞋去学校。放学回家，吃完晚饭，休息一会儿开始写作业。今天只有语文作业，但老师布置了 20 道相同的抄写题，小明没有办法，只有一道一道地做，直到全部做完，再去睡觉。

从上面的例子不难看出，小明在这一天中的行为方式具有一些明显特征。如“早上起床、洗漱、吃早点”和“放学回家、吃晚饭、休息、写作业”等一系列的活动是按部就班的，按顺序一步步地完成的。这是一种“顺序”特征。

“走到门口，准备像往常一样穿上皮鞋，但想到今天有体育课，就穿了运动鞋去学校”，小明在穿鞋的问题上做了个判断，根据当前情况决定穿什么鞋。如果没有体育课，他会穿皮鞋。这反映出一种“选择”特征。

“老师布置了 20 道相同的抄写题，小明没有办法，只有一道一道地做，直到全部做完”，这里明显反映出“重复”和“循环”的特征。

生活中还可以举出很多类似的例子，虽然具体情况不同，但在处理流程上都表现出“顺序”“选择”或“循环”的特征。可以这样说，任何问题都是按照某些特定的处理模式来进行处理的。人类按照这样的方式处理问题，计算机其实和人类处理问题的方式一样，但计算机要对这些处理方式进行具体、精确的描述，形成“算法”，并让处理器按照算法执行，从而完成问题的处理。

3.1.2 算法的结构化表示

问题的处理一般表现为顺序、选择和循环 3 种模式，算法要采用相应的方法将各种处理模式表示出来，常用的表示方法有自然语言、流程图、伪代码等方式。

本节只介绍算法的流程图表示方式，因为流程图通俗易懂，能很好地反映算法的“结构性”。流程图有两种：一种是传统流程图；另一种是 N-S 流程图。下面分别介绍。

1. 传统流程图

传统流程图用一系列图形、流程线来描述算法。传统流程图的基本图形元素如图 3-1 所示。

图 3-1　传统流程图的基本图形元素

应用上述图形元素，顺序、选择和循环表示的 3 种处理模式如图 3-2 所示。

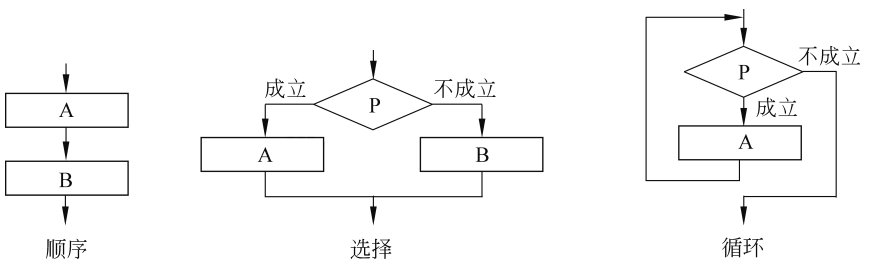

图 3-2　传统流程图表示的 3 种处理模式

【例 3-2】 将例 3-1 的处理流程用传统流程图表示。

根据前面的分析，例 3-1 所举的生活中的例子能较容易地用传统流程图表示，如图 3-3 所示。

从图 3-3 可以看到，用传统流程图表示的算法逻辑清楚、易懂，结构性较好。但缺点是，一旦算法复杂，则绘制比较麻烦，尤其是流程线若存在过多，会导致算法的结构清晰度变差。下面介绍一种结构性更好的流程图表示形式。

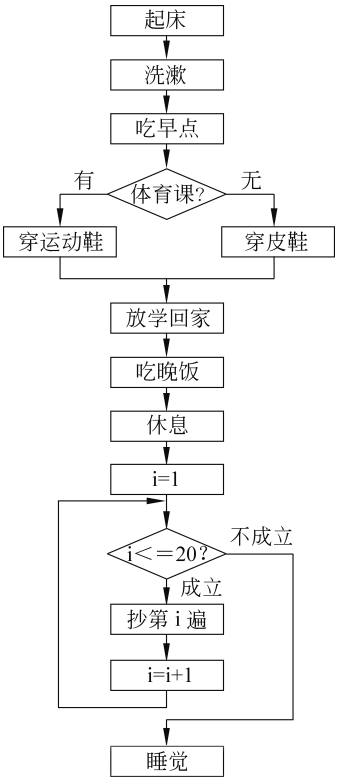

图 3-3　例 3-1 的传统流程图

2. N-S 流程图

1973 年，美国的计算机科学家 I.Nassi 和 B.Shneiderman 提出了一种新的流程图形式，在这种流程图中把流程线完全去掉了，全部算法写在一个矩形框内，在框内还可以包含其他框，即由一些基本的框组成一个较大的框。这种流程图称为 N-S 流程图（以两人名字的头一个字母组成），这种流程图能非常清晰地反映算法的结构性。图 3-4 用 N-S 流程图表示顺序、选择和循环结构。

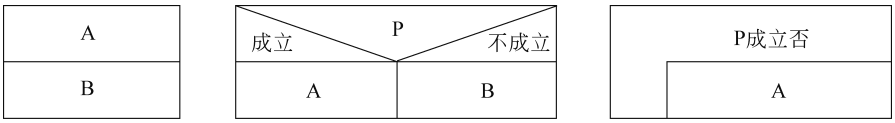

图 3-4　N-S 流程图表示的 3 种处理模式

【例 3-3】 将例 3-1 的处理流程用 N-S 流程图表示成如图 3-5 所示的形式。

从图 3-5 可以看出，相对于传统流程图，用 N-S 流程图表示的算法结构更清晰，更能体现算法的结构性质。

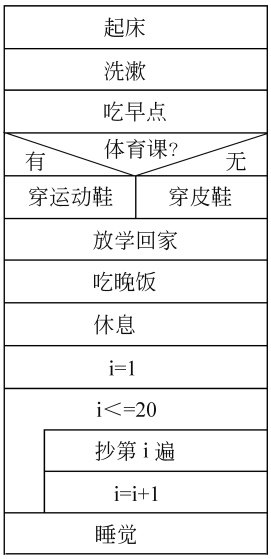

图 3-5　例 3-1 的 N-S 流程图

3.1.3　算法的语言表示

流程图能形象地以“结构化”的方式展示算法，但遗憾的是，无论是传统流程图，还是N-S 流程图，它们表示的算法计算机都不能识别执行，因为计算机只能理解用计算机语言（程序设计语言）所表示的算法。流程图已经很好地以结构性的方式将算法表示出来，只要掌握了某种程序设计语言的语法，就能很方便地用该种语言将算法进行精确的表示。本书讲述的是 Python 语言，所以都是用 Python 语言来表示算法。在各种处理结构中，顺序结构是最简单、也是最常用的结构形式，前面章节中的大部分例子都是顺序结构的，下面从流程图到语言算法的实现角度再对顺序结构加以说明。

【例 3-4】　任意输入两个整数到变量 x、y 中，将 x、y 中的数据交换并输出。

经简单分析，可画出如图 3-6 所示的 N-S 流程图。

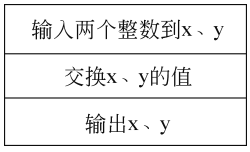

图 3-6　例 3-4 的 N-S 流程图

按照该流程图，利用 Python 语言实现的程序如下：

```
#liti3-4-1.py
x,y = eval(input('任意输入两个整数(用逗号间隔):'))
x,y = y,x                          #交换 x、y 的值
print('x={0},y={1}'.format(x,y))
```

这种实现方式代码很精炼，但该例也可以用如下方式实现。

```
#liti3-4-2.py
x,y = eval(input('任意输入两个整数(用逗号间隔):'))
t = x                              #用变量 t 来交换 x、y 的值
x = y
```

```
y = t
print('x={0},y={1}'.format(x,y))
```

两种方法各有特点，读者可自行理解、比较，并上机练习加以体会。

从本节可以看出，算法无论是用流程图、编程语言还是其他的方式来表示，核心都是结构化的思想，即用各种流程控制结构来进行算法的设计和细化。具体地说，就是在编程前先对问题进行分析，得出问题的基本解决方案模型，再按结构化的思想对解决方案进行算法的组织和设计，直到给出算法的完整处理步骤，最后用编程语言实现该算法。

顺序结构是一种最简单的流程控制结构，本章后面的内容将依次对选择结构和循环结构进行详细介绍。

3.2 选 择 结 构

选择结构也称为分支结构，用于实现根据条件处理不同的问题。选择结构可以分为双分支、单分支、多分支以及嵌套的分支结构等类型。

3.2.1 双分支结构

双分支结构是一种最常见的分支结构，N-S 流程图如图 3-7 所示。

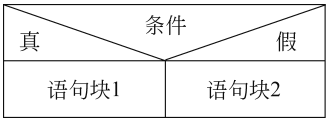

图 3-7 双分支结构的 N-S 流程图

对应的 Python 语句为

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

执行双分支语句时，当条件为 True 时执行语句块 1，条件为 False 时执行语句块 2；语句块 1 和语句块 2 只能执行一个，执行完某一个即结束了该分支结构的执行。注意，if 和 else 后面都要加上冒号，语句块 1 和语句块 2 两个分支都要有一致的缩进。

条件表达式是实现选择结构或循环结构不可缺少的部分，条件表达式可以用关系运算符、逻辑运算符、成员运算符、身份运算符等来实现，如 1＜x＜10、a＝＝b、x＞y and a!＝b、x in ls、a is b 等。注意，赋值运算符＝不能出现在条件表达式中，不能和＝＝运算符混淆。

【例 3-5】 输入成绩 grade，分两种情况考虑，低于 60 分输出“不合格”，60 分以上输出“合格”。

根据问题描述，可以得出要用双分支的选择结构来表示处理步骤，对应的 Python 程序如下。

```
#liti3-5-1.py
grade = eval(input('输入成绩:'))
```

```
if grade<60:
    print('不合格')
else:
    print('合格')
```

程序运行结果如下：

```
输入成绩:80
合格
```

再次运行该程序：

```
输入成绩:50
不合格
```

在 Python 中，双分支结构还可以用一种更简洁的形式表示。Python 提供了一个三元条件运算符，其构造的表达式可以实现与双分支结构相似的效果。语法为

```
<表达式 1> if <条件> else <表达式 2>
```

当条件为真时，该三元表达式的值取表达式 1 的值，否则取表达式 2 的值。这种表达式很容易产生误解，认为就是选择语句。其实在此处的 if…else…不是用于表示选择结构语句，而是作为一种特殊的三元运算符，用于构造一种特殊的表达式。因此，例 3-5 也可以表示如下：

```
#liti3-5-2.py
grade = eval(input('输入成绩:'))
print('不合格' if grade<60 else '合格')
```

可以通过下面的例子进一步理解三元条件表达式：

```
>>> a = 2
>>> a if a!=0 else '非法'
2
>>> a = 0
>>> a if a!=0 else '非法'
'非法'
>>> x = eval(input('x='))
x=3
>>> y = eval(input('y='))
y=5
>>> x if x>y else y          #等价于 max(x,y)
5
```

3.2.2 单分支结构

单分支结构是一种特殊的选择结构，N-S 流程图如图 3-8 所示。

对应的 Python 语句为

```
if <条件>:
    语句块
```

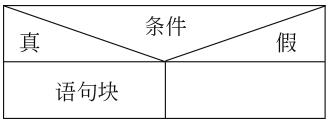

图 3-8　单分支结构的 N-S 流程图

执行单分支语句时，当条件为 True 时执行语句块，条件为 False 时什么都不执行，直接结束该单分支语句。

【例 3-6】 输入成绩 grade，对成绩分如下 4 种情况考虑。若成绩为[0，60)，输出“差”；成绩为[60，70)，输出“中”；成绩为[70，80)，输出“良”；成绩为[80，100]，输出“优”。

该例可用多种方法解决，此处用单分支结构解决，学了后面的多分支和分支嵌套后，对该例再分别用其他方法实现，并在不同方法间进行比较，总结不同结构的特点。该例用单分支结构实现如下：

```
#liti3-6.py
grade = eval(input('输入成绩:'))
if 0<=grade<60:
    print('差')
if 60<=grade<70:
    print('中')
if 70<=grade<80:
    print('良')
if 80<=grade<=100:
    print('优')
```

程序运行结果如下：

```
输入成绩:50
差
```

多次运行程序，输入位于不同区间的分数，从结果看，可验证该程序是正确的。但仔细分析，程序存在一个问题。如上面程序运行所示，输入成绩 50，输出“差”。因为成绩满足条件“0＜＝grade＜60”，所以执行语句 print('差')。但输出“差”后，程序并没有结束，还会依次执行后面的 3 个单分支语句。这 3 个单分支语句的执行是没有必要的，因为当前“0＜＝grade＜60”的值为 True，则条件“60＜＝grade＜70”“70＜＝grade＜80”“80＜＝grade＜＝100”的值肯定是 False。但为什么后面 3 个 if 语句也会执行呢？因为这 4 个单分支 if 语句是各自独立的、并列的语句，它们之间是一种顺序结构，无论自身条件是真还是假，对别的语句都不构成任何影响。即不论输入的是什么数，这 4 个 if 语句都会无条件执行。显然这是一种不好的现象，程序运行中做一些不必要的运算，不但浪费了处理器的时间，同时也降低了程序的运行速度。所以程序的好坏不能只从结果来看，还要从程序的结构是否合理、精巧来判断。下面的多分支结构能很好地解决这个问题。

3.2.3　多分支结构

多分支结构的语法格式如下：

```
if <条件 1>:
    语句块 1
elif <条件 2>:
    语句块 2
⋮
elif <条件 n-1>:
语句块 n-1
[else:
    语句块 n]
```

从多分支结构的语法可以看出，当多分支结构执行时，会从上到下对条件依次判断，一旦碰到成立的条件，则转去执行对应的分支语句块，然后结束整个多分支结构的执行，不会对后面的条件再进行多余的判断。else 子句是可选的，若 else 子句存在，则当前面的条件都不成立时，执行 else 下面的语句块。因此，从结构特点和执行效率来看，多分支结构是实现多种情况(3 个以上)处理的最佳方式。

【例 3-7】 对例 3-6 用多分支结构实现。

用多分支结构实现的程序代码如下：

```
#liti3-7-1.py
grade = eval(input('输入成绩:'))
if 0<=grade<60:
    print('差')
elif 60<=grade<70:
    print('中')
elif 70<=grade<80:
    print('良')
elif 80<=grade<=100:
    print('优')
```

上述程序以多分支的方式很好地解决了该问题，读者应将该例和例 3-6 的实现进行比较，仔细体会两种方式的不同。

该程序还可以进一步优化。因为多分支是一个完整的独立结构，其中的多个条件是按先后依次判断的，因此条件可以简化。例如，“60＜＝grade＜70”可以改为“grade＜70”，“70＜＝grade＜80”可以改为“grade＜80”；最后一个 elif 分支可以用 else 取代，不需要再判断，因为若前面 3 个条件都不成立，则分数一定是在 80 以上的。因此，程序可修改如下：

```
#liti3-7-2.py
grade = eval(input('输入成绩:'))
if 0<=grade<60:
    print('差')
elif grade<70:
    print('中')
elif grade<80:
```

```
    print('良')
else:
    print('优')
```

程序代码简洁，提高了可读性，但经过测试，发现修改后的程序存在问题。

```
输入成绩:-5
中
输入成绩:110
优
```

修改后的程序代码虽然简洁了，但忽视了对边界数据的检测，导致程序不能有效处理小于 0 或大于 100 的数，从而出现了错误的结果。程序不正确，再怎么追求代码的简洁都是没有任何意义的！难道这个程序只能采用原来的写法，不能“鱼与熊掌兼得”吗？其实，只要先对数据的合法性进行判断，程序的正确性和简洁性是可以兼得的。下面要介绍的嵌套分支结构就能很好地解决这个问题。

3.2.4 嵌套分支结构

一个分支结构完全包含了另一个分支结构，就形成了分支结构的嵌套。嵌套分支结构的一般语法如下：

```
if <条件 1>:
    ⋮
    if <条件 2>:
        语句块 2
    [else:
        语句块 3]
    ⋮
else:
    ⋮
    if <条件 4>:
        语句块 4
    [else:
        语句块 5]
    ⋮
```

上述只是嵌套分支结构的一般情况，实际中的嵌套情况可以根据具体问题灵活变化，if 分支或 else 分支中不一定都要有嵌套，并且嵌套的层数没有限制，嵌套的结构里面还可以继续嵌套。

【例 3-8】 对例 3-7 用嵌套的分支结构实现既强健又简洁的程序。

在 3.2.3 节中，修改后的程序缺少对边界数据的检测，导致程序运行不稳定。解决的办法是先安排一个双分支结构对输入数据的合法性进行判断，若合法则在内部嵌套一个多分支结构对每种情况进行处理，否则输出数据非法的提示信息。相应程序如下：

```
#liti3-8-1.py
grade = eval(input('输入成绩:'))
if 0<=grade<=100:
    if grade<60:
        print('差')
    elif grade<70:
        print('中')
    elif grade<80:
        print('良')
    else:
        print('优')
else:
    print('成绩错误')
```

程序运行结果如下：

```
输入成绩:-5
成绩错误
输入成绩:110
成绩错误
输入成绩:88
优
```

上述程序用带有“0＜＝grade＜＝100”条件的选择结构对数据的合法性进行检测，若合法则在 if 分支中安排一个多分支结构继续处理。由于外部限定了合法数据的范围，内部嵌套的多分支结构的条件可以进一步简化为“grade＜60”“grade＜70”“grade＜80”等。这样的解决方案，不但保证了程序的正确性，同时程序代码也不失简洁、结构清晰的特点。

解决该问题的思路稍做改变，还可以写出以下结构不同、但功能一样的程序：

```
#liti3-8-2.py
grade = eval(input('输入成绩:'))
if 0<=grade<=100:
    if grade<60:
        print('差')
    else:
        if grade<70:
            print('中')
        else:
            if grade<80:
                print('良')
            else:
                print('优')
else:
    print('成绩错误')
```

上面使用的是 if 语句的嵌套，下面采用的是多分支的 if 语句，结构更清晰、更容易

理解。

```
#liti3-8-3.py
grade = eval(input('输入成绩:'))
if grade<0 or grade>100:
    print('成绩错误')
elif grade<60:
    print('差')
elif grade<70:
    print('中')
elif grade<80:
    print('良')
else:
    print('优')
```

分支结构是一种常用且重要的处理控制结构,可分为单分支、双分支、多分支、嵌套分支等多种类型。初学者应多采用一题多解的方式,因为该方式能使初学者快速、深入把握各种算法处理结构,并达到灵活运用的目的。

3.2.5 选择结构综合举例

【例 3-9】 任意输入 3 个整数 a、b、c,按从大到小降序排列。

先比较 a 和 b,保证 a≥b;再比较 a 和 c,保证 a≥c,此时 a 是三者中的最大数;最后比较 b 和 c,保证 b≥c。经过这 3 步处理,a、b、c 已降序排列。这 3 步对应到 3 个单分支结构,相应程序如下:

```
#liti3-9-1.py
a = int(input('a='))
b = int(input('b='))
c = int(input('c='))
if a<b:
    a,b = b,a
if a<c:
    a,c = c,a
if b<c:
    b,c = c,b
print("降序结果:",a,b,c)
```

程序运行结果如下:

```
a=3
b=8
c=5
降序结果: 8 5 3
```

上述程序采用了 Python 语言特有的交换方式，代码比较简练。该程序还可以进一步简化。在 a、b、c 基础上创建列表[a，b，c]，利用内置函数 sorted()对该列表进行降序排列，返回一个降序排列的列表，再对该降序列表进行多重赋值操作，将元素依次赋给变量 a、b、c 并输出。相应程序如下：

```
#liti3-9-2.py
a = int(input('a='))
b = int(input('b='))
c = int(input('c='))
a,b,c = sorted([a,b,c],reverse=True)
print('降序结果:',a,b,c)
```

可以看到，程序的 Python 味道越来越浓，代码也越来越精练。这是否已是该问题的终结版了？回答是否定的，还有更简练的，程序如下：

```
#liti3-9-3.py
a,b,c = sorted(eval(input('输入 3 个整数(逗号间隔):')),reverse=True)
print('降序结果:',a,b,c)
```

程序运行结果如下：

```
输入 3 个整数(逗号间隔):5,8,3
降序结果: 8 5 3
```

此处只有两行代码。input()函数返回一个由逗号间隔的 3 个整数构成的字符串，如"3,8,5"。eval()函数相当于将字符串两端的引号去掉，得到表达式的原本意义。Python 中，表达式 3,8,5 表示一个元组，则 eval("3,8,5")得到元组(3，8，5)。再用 sorted()函数对该元组进行降序排列 sorted((3，8，5),reverse＝True)，得到列表[8，5，3]。最后将排序得到的列表进行多重赋值，将其中的元素依次赋给变量 a、b、c，得到所需结果。

【例 3-10】 输入年份，判断是否为闰年。符合以下两个条件中的任何一个都是闰年：①能被 400 整除；②能被 4 整除但不能被 100 整除。

分析角度不同，可以写出不同的程序。

方法一：使用多分支结构，程序代码如下：

```
#liti3-10-1.py
y = int(input('输入年份:'))
if (y%400==0):
    print('是闰年')
elif (y%4==0 and y%100!=0):
    print('是闰年')
else:
    print('不是闰年')
```

读者也可以将该问题用嵌套的分支结构实现，多分支结构和嵌套分支结构在某些情

况下是等价的。

方法二：使用双分支结构，程序代码如下：

```
#liti3-10-2.py
y = int(input('输入年份:'))
if (y%4==0 and y%100!=0) or (y%400==0):
    print('是闰年')
else:
    print('不是闰年')
```

将两个条件用逻辑运算符 or 连接起来，构成一个完整的判断闰年的条件表达式，程序代码就简短了。

其实还有别的方法，例如通过内置模块 calendar 中的 isleap()函数来实现闰年的判断，程序更易编写，感兴趣的读者可以自行尝试。

【例 3-11】 输入三角形的 3 条边，判断其构成的是等边三角形、等腰三角形、直角三角形，还是普通三角形。

首先对输入的 3 边进行判断，能否构成三角形，能则再判断是 4 种三角形里面的哪一种，否则输出错误提示。相应程序如下：

```
#liti3-11-1.py
a,b,c = eval(input('输入 3 条边(逗号间隔):'))
if a+b>c and a+c>b and b+c>a:
    if a==b==c:
        print('等边三角形')
    elif a==b or a==c or b==c:
        print('等腰三角形')
    elif a**2+b**2==c**2 or a**2+c**2==b**2 or b**2+c**2==a**2:
        print('直角三角形')
    else:
        print('普通三角形')
else:
    print('不能构成三角形')
```

上述程序能全面、正确地解决该问题，但问题就是有些条件太长了，程序看上去不简洁。有没有办法将条件变短呢？其实只要对输入的 3 边先进行排序(升序)，后面的条件就可以大大简化，为此程序的第一条语句可以改为 a,b,c = sorted(eval(input('输入 3 条边(逗号间隔)：')))。因为 a、b、c 升序排列了，则判断能否构成三角形只须判断 a+b>c 即可，且 a、b、c 升序排列后，a+c>b 和 b+c>a 是绝对成立的。类似地，判断等边三角形只须 a==c；判断等腰三角形只须 a==b or b==c；判断直角三角形只须 a**2+b**2==c**2。因此，修改后的程序如下：

```
#liti3-11-2.py
a,b,c = sorted(eval(input('输入 3 条边(逗号间隔):')))
if a+b>c:
```

```
        if a==c:
            print('等边三角形')
        elif a==b or b==c:
            print('等腰三角形')
        elif a**2+b**2==c**2:
            print('直角三角形')
        else:
            print('普通三角形')
    else:
        print('不能构成三角形')
```

经过优化，程序变得简洁，可读性也更好了，这样的程序不也反映出一种美吗？

程序的确看起来清爽了很多，但这个程序是否已完美地解决了这个问题呢？仔细分析，还是能发现美中不足的地方。例如，若输入的 3 条边能构成一个等腰直角三角形，按照上面的程序却只能输出“等腰三角形”的判断，漏掉了“直角”这个重要特征，这显然不合适，输出“等腰直角三角形”的判断才是最适宜的。但当前这个程序做不到，如何改进呢？感兴趣的读者可以自行思考。

所以，编程不能只满足于一蹴而就，后期对程序的完善也非常重要，这一点初学者尤其要注意。只有对程序结构、算法的孜孜以求，才能进一步激发编程的兴趣，发现编程的魅力所在，并不断提高自身的程序设计水平。

3.3 循环控制结构

循环即往复回旋，指事物周而复始地运动或变化。在程序设计中，通过循环，可以使一组语句重复执行多次。例如，要计算 100 以内的偶数和，可以使用一个变量来存储偶数的和，假设使用变量 sum 来保存其和，在求和之前使 sum 的值为 0，即 sum＝0。假设使用变量 i 来遍历[2,100)的偶数，i 每遍历一个偶数，使语句 sum＝sum＋i 执行一次；当 i 遍历完[2,100)的偶数后，sum 中就保存了 100 以内的偶数和。在 Python 语言中通过 for 或 while 可以实现循环控制。

3.3.1 while 循环语句

在 Python 中，while 循环语句的语法格式如下：

```
while 循环条件:
    循环体语句
```

只要循环条件成立，while 语句中的语句序列就会一直执行，这个语句序列叫作 while 语句的循环体语句。

while 循环语句的执行过程如下。

(1) 计算循环条件,若条件为真,则执行步骤(2);否则结束循环。

(2) 执行循环体语句,执行完后继续执行步骤(1)。

while 循环语句的执行过程可以用图 3-9 表示。

【例 3-12】 用 while 循环语句,编程求 100 以内(不包括 100)的偶数和。

分析:使用了 sum 来保存偶数和,使用了变量 i 来遍历 2～98 的偶数。变量 sum 和 i 在进入循环之前都要对其进行初始化,即在没有求和之前,应使 sum 的值为 0。100 以内最小的非零偶数为 2,因此 i=2,即表示 i 从最小的 2 开始遍历,一直要遍历到 98,所以循环的结束条件是 i<100。每循环一次累加一个偶数,并计算出下一次要累加的偶数。程序的算法流程图如图 3-10 所示,程序代码如下:

```
#liti3-12.py
sum=0
i=2
while i<100:
    sum+=i
    i=i+2
print("2+4+…+%d=%d"%(i-2,sum))
```

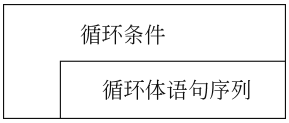

图 3-9 while 循环语句的执行过程

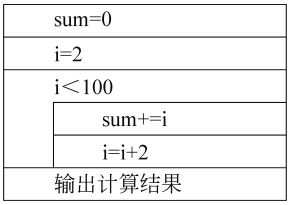

图 3-10 例 3-12 的算法流程图

程序运行结果如下:

```
2+4+…+98=2450
```

在本例中,请读者思考如下 3 个问题。

(1) while 后面的循环条件能否改为 i<=100?

(2) 在 print 输出中,为什么用 i-2,而不是 i?

(3) 在程序 liti3-12.py 中使用循环变量 i,从 2 遍历到 98,如果要让循环变量 i 从 98 开始遍历到 2,上述程序应如何修改?

【例 3-13】 小明现有 10000 元准备存入银行,进行理财。目前银行三年以上的定期年利率是 2.75%,假设利率不变,可以利滚利,问至少存多少年后小明的账户余额可以翻倍?

使用变量 rate、bal、year 来分别存储利率、账户余额和年份,在进入循环前给它们赋题目要求的初值,当账户余额小于目标值(20000)时,计算下一年份的余额,并累加年份,循环结束后输出所求的年份和账户余额。程序的算法流程图如图 3-11 所示,程序代码如下:

```
#liti3-13.py
rate=2.75
bal=10000
year=0
while bal<20000:
    year=year+1
    bal=bal+bal*rate/100
print("year=%d,bal=%.2f"%(year,bal))
```

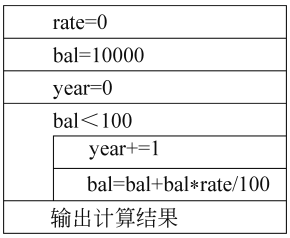

图 3-11　例 3-13 的算法流程图

程序运行结果如下：

```
year=26,bal=20245.46
```

在本例中，请读者思考如下两个问题。

（1）程序中的循环条件能否改为 bal≥20000？如果改为 bal≥20000，while 的循环体语句执行了多少次？

（2）year 的初值为什么是 0，能否是 1？

总结上述两个例题，使用 while 循环语句设计程序，需要注意以下事项。

（1）在进入循环前要给循环控制变量赋初值，如例 3-12 中的 i 和例 3-13 中的 bal。一般来说，循环变量的初值是循环开始的值，循环条件是用循环变量与终值进行比较。

（2）在循环体内至少要有一条语句能够改变循环变量的值，使之趋向循环结束条件，否则，循环将永远不会结束，把永远不会结束的循环称为“死循环”。例如，把例 3-12 中的程序代码修改如下：

```
sum=0
i=2
while i<100:
    sum+=i
print("2+4+…+%d=%d"%(i-2,sum))
```

在 Python IDLE 环境下运行上述程序，可以看到光标一直在闪烁，结果出不来，就是因为上述程序有“死循环”。此时可按 Ctrl＋C 组合键强行终止程序的运行，并修改程序。

（3）在格式上，循环条件后面一定要有冒号“：”，循环体语句一定要进行缩进，循环结束后的第一条语句一定要与 while 对齐，否则程序运行时会出错。

3.3.2　for 循环语句

Python 中的 for 循环语句用于遍历可迭代对象中的每个元素，并根据当前访问的元素做数据处理，其语法格式如下：

```
for 变量 in 可迭代对象:
    循环体语句
```

for 循环语句的执行过程是依次取可迭代对象中的每个元素的值，如果有，取出赋给

变量，执行循环体语句，否则循环结束。

for 循环语句的执行过程可用图 3-12 表示。

【例 3-14】 用 for 循环语句，编程求 100 以内(不包括 100)的偶数和。

和例 3-12 一样，使用变量 sum 来保存偶数和的值，在求和之前赋初值为 0，即 sum＝0，再用循环变量 i 来遍历 2～100 的偶数(不包括 100)，每循环遍历一次，累加一个偶数。程序的算法流程图如图 3-13 所示，程序代码如下：

```
#liti3-14.py
sum=0
for i in range(2,100,2):
    sum+=i
print("2+4+…+%d=%d"%(i,sum))
```

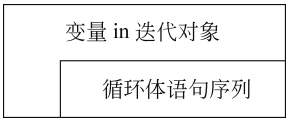

图 3-12 for 循环语句的执行过程

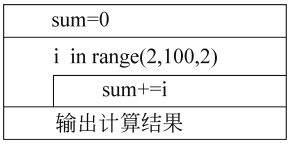

图 3-13 例 3-14 的算法流程图

程序运行结果如下：

```
2+4+…+98=2450
```

在使用 for 循环语句时，如果需要遍历一个数列中的所有数字，则可以用 range()函数生成一个可迭代对象。程序中用了 range(2,100,2)，表示的就是从 2 开始，到 100 终止(不包含 100)，步长为 2 的迭代对象。

【例 3-15】 已知华氏温度与摄氏温度之间的转换关系：C＝(5/9)*(F－32)。其中，F 为华氏温度，C 为摄氏温度。编写程序，分别显示华氏温度 0，10，20，…，100 与摄氏温度的对照表。

该程序的算法非常简单，流程图省略，程序代码如下：

```
#liti3-15.py
print("华氏温度\t 摄氏温度")
for F in range(0,101,10):
    C=5/9*(F-32)
    print("%d\t\t%0.2f"%(F,C))
```

程序运行结果如下：

```
华氏温度    摄氏温度
0           -17.78
10          -12.22
20          -6.67
30          -1.11
40          4.44
50          10.00
```

```
60   15.56
70   21.11
80   26.67
90   32.22
100  37.78
```

程序 print()函数使用了转义字符'\t',表示到下一制表位,一个制表位表示 8 字节宽度,常用于控制输出对齐,详见第 6 章。

【例 3-16】 已知 list1=["Python","C++ ","Basic","Java"],用 for 循环实现,一行输出一个元素。

程序代码如下:

```
#liti3-16.py
list1=["Python","C++","Basic","Java"]
for ls in list1:                    #遍历列表 list1 中的每个元素
    print(ls)
```

程序运行结果如下:

```
Python
C++
Basic
Java
```

使用 for 循环语句设计程序,需要注意以下事项。

(1) 可迭代对象可以是列表、元组、字符串、集合和字典等序列数据,也可以是 range 等可迭代函数。如果是字典,每次遍历时获取的是元素的键,通过键可以再获取元素的值,代码如下:

```
dict1={1:"Python",2:"C++",3:"Basic",4:"Java"}
for d in dict1:
    print(d,":",dict1[d])
```

程序运行结果如下:

```
1 : Python
2 : C++
3 : Basic
4 : Java
```

(2) 在格式上,for 后面一定要有冒号(:),循环体语句一定要进行缩进,循环结束后的第一条语句要与 for 对齐,否则程序运行时会出错。

3.3.3 break 语句

break 语句用于跳出 for 循环或 while 循环,转到循环后的第一条语句去执行。如果

break 用在多重循环中，则 break 语句只能跳出它所在的本层循环。

其语法格式如下：

```
break
```

【例 3-17】 从键盘输入一个正整数 n，判断 n 是否是素数。

素数是指只能被 1 和它本身整除的数。可以使用反证法，用变量 i 遍历 2～n－1，若 n 能被某个 i 整除，则说明 n 不是素数，提前结束循环，此时 i 一定小于 n，输出“不是素数”，否则输出“是素数”。程序代码如下：

```
#liti3-17.py
n=int(input("请输入一个正整数:"))
for i in range(2,n+1):
    if n%i==0:                  #如果 n 能被 i 整除
        break                   #说明 n 不是素数,跳出循环,此时 i 小于 n
if i<n:
    print("%d 不是素数!"%n)
else:
    print("%d 是素数!"%n)
```

程序运行结果如下：

```
请输入一个整数:16
16 不是素数!
```

再运行一次程序：

```
请输入一个整数:37
37 是素数!
```

上述算法是让 n 去除以 2～n－1 的每个整数，如果都不能整除，说明 n 就是素数。该算法中当 n 是一个非常大的素数时，要执行 n－1 次循环，效率很低。为了提高效率，必须减少循环的次数。是否可以不除到 n－1，例如除到 n/2 或$\sqrt{n}$即可判定 n 是否是素数，如果可以，读者自行修改程序。

3.3.4 continue 语句

continue 语句用于结束本次循环并开始下一次循环，对于多重循环情况，continue 语句作用于它所在的本层循环。

【例 3-18】 阅读下列程序代码。

程序代码如下：

```
#liti3-18.py
sum=0
while True:
```

```
    n=eval(input('请输入一个整数(输入 0 结束程序):'))
    if n==0:
        break
    if n%3!=0:
        continue
    sum+=n
print("sum=%d"%sum)
```

运行该程序时,若依次输入 15、20、35、40、45、50 和 0,则最后输出 sum 等于多少呢?

在上面的程序中,循环条件设置为 True,通常称这种循环为“永真循环”,即不可以通过条件不成立退出循环。对于这种“永真循环”,在循环体中必然包含 break 语句退出循环,否则将导致死循环,程序无法正常退出。该程序中,如果输入的值为 0,则退出循环,如果输入的值不是 3 的倍数,则结束本次循环,进行下一次循环。因此该程序的运行结果为 60。

3.3.5 else 语句

在 Python 语言中,for 循环和 while 循环可以有 else 分支语句。当 for 循环已经遍历完可迭代对象中的所有元素或 while 循环的条件为 False 时,就会执行 else 分支中的语句。也是就说,在 for 循环或 while 循环中,只要不是执行到 break 语句而退出循环的,如果有 else 语句,就要执行 else 分支语句块。

带 else 分支的 for 循环的语法格式如下:

```
for 变量 in 可迭代对象:
    循环体语句
else:
    分支语句块
```

带 else 分支的 while 循环的语法格式如下:

```
while  循环条件:
    循环体语句
else:
    分支语句块
```

【例 3-19】 从键盘输入一个正整数,判断该数是否是素数。用带 else 分支的语句改写例 3-17 中的程序代码。

程序代码如下:

```
#liti3-19.py
n=int(input("请输入一个正整数:"))
for i in range(2,n):
    if n%i==0:
        print("%d 不是素数!"%n)
```