

## 第5章 复杂逻辑实现——程序控制结构

### || 学习目标

- (1) 理解分支逻辑
- (2) 掌握 Python 中分支结构的实现:单分支、双分支、多分支、分支嵌套
- (3) 理解循环逻辑
- (4) 掌握循环结构的实现
- (5) 掌握 break 和 continue 语句
- (6) 熟悉循环的 else 子句

程序控制结构是编程中用于控制程序执行流程的机制,它们决定了程序中语句的执行条件和顺序。基本的程序控制结构有顺序结构、分支结构和循环结构。任意复杂逻辑均可由这3种基本结构组合实现。

### || 5.1 顺序结构

顺序结构是指按语句出现的先后顺序依次执行的程序结构,其逻辑如图 5-1 所示,按照语句的排列顺序从上到下逐条执行。



图 5-1 顺序结构流程图

第4章中的实例 4.6、4.7、4.8 均是顺序结构。

## 5.2 分支结构

在例 3.1 计算 BMI 值时,程序可以根据不同的 BMI 值给出不同的健康建议。像这种需要分情况处理的情况,就需要用分支结构实现。分支结构又称为选择结构,它根据条件来选择执行路径。根据分支路径的不同,有单分支结构、双分支结构和多分支结构。

### 5.2.1 单分支结构

单分支结构如图 5-2 所示,条件为真时执行语句块,条件为假时不做处理。

单分支结构的语法为:

```
if 条件表达式:
    语句块
```

**例 5.1** 输入两个整数,按从小到大的顺序输出(eg5\_1\_compareData.py)。

分析思路(图 5-3):

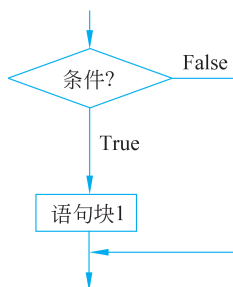


图 5-2 单分支结构

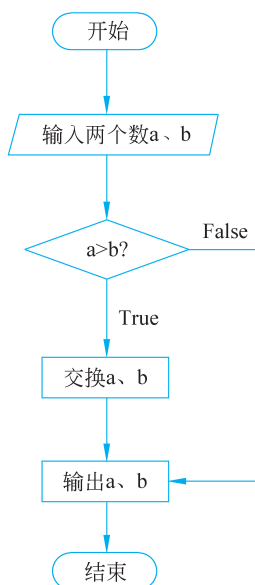


图 5-3 数据由小到大输出流程图

参考代码:

```
a = int(input("请输入第一个数: "))
b = int(input("请输入第二个数: "))
if a > b:
    a, b = b, a
print("从小到大的顺序为: ", a, b)
```

Python 程序控制结构中的条件表达式可以是任意表达式,只要结果不是 False、0(或 0.0、0j)、空值、None、空列表、空元组、空集合、空字典、空字符串、空 range 对象或其他空迭代对象,Python 解释器均认为其与 True 等价。条件表达式判断是否相等,需要使用“==”。



语句块中可以是一条语句,也可以是多条语句,多条语句的缩进必须一致。

例如使用整数作为条件表达式:

```
if 3:
    print(5)
```

输出结果为 5。

使用列表作为条件表达式,如果列表有内容,则认为条件表达式为 True;如果列表为空,则认为条件表达式为 False。例如:

```
a=[1,2,3]
if a:
    print(a)
```

输出结果为: [1, 2, 3]。

```
a=[]
if a:
    print(a)
else:
    print('empty')
```

输出 empty。

### 5.2.2 双分支结构

双分支结构如图 5-4 所示,条件表达式为真时执行语句块 1,否则执行语句块 2。

双分支结构的语法如下:

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

**例 5.2** 输入一个年份,判断其是不是闰年。(eg5\_2\_leapYear.py)

闰年的条件: 年份能被 4 整除但不能被 100 整除或者能被 400 整除。假设年份用  $y$  表示,则闰年的条件为  $(y \% 4 == 0 \text{ and } y \% 100 != 0) \text{ or } y \% 400 == 0$ 。

参考代码如下:

```
y=int(input("请输入年份"))
if ((y%4==0 and y%100!=0) or y%400==0):
    print(f"{y}年是闰年")
else:
    print(f"{y}年不是闰年")
```

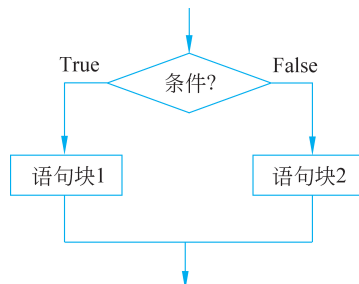


图 5-4 双分支结构

### 5.2.3 多分支结构

当要处理的情况多于两种时,可以用多分支结构或分支结构的嵌套来实现。多分支结构如图 5-5 所示。

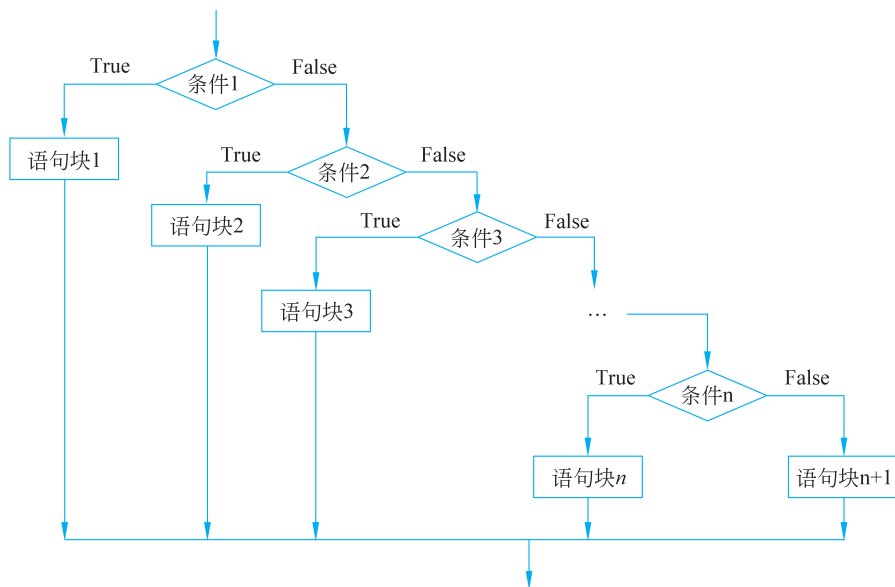


图 5-5 多分支结构

多分支结构的语法格式为：

```

if 条件表达式 1:
    语句块 1
elif 条件表达式 2:
    语句块 2
...
elif 条件表达式 n:
    语句块 n
else :
    语句块 n+1
    
```

**例 5.3** 猜数字游戏：随机产生一个数 num，用户输入一个数 guess，如果两个数相等，则输出“中奖了!”；如果  $guess < num$ ，则输出“猜小了”；如果  $guess > num$ ，则输出“猜大了”。(eg5\_3\_guessNum.py)

```

import random
num=random.randint(1,10)
guess=int(input("please guess a number"))
if guess==num:
    print("中奖了!")
elif guess<num:
    print("猜小了")
else:
    print("猜大了")
    
```

第 2 行 `random.randint(1,10)` 生成一个  $[1,10]$  的随机整数 num，第 3 行输入一个整数 guess，第 4 行判断 guess 与 num 是否相等。如果相等，则输出“中奖了!”程序结束；否则再判断 guess 是否小于 num，如果小于，则输出“猜小了”，程序结束；否则输出“猜大了”。

**例 5.4** 朋友们一起坐地铁出游，输入人数和要乘坐的站数，输出应支付的金额。假设地铁采用按里程计费的方式，具体收费标准如下：



- 起步价 3 元,可乘坐 6 千米。
- 6~12 千米,加收 1 元,票价为 4 元。
- 12~22 千米,再加收 1 元,票价为 5 元。
- 22~32 千米,再加收 1 元,票价为 6 元。
- 32 千米以上,均为 7 元。

思路分析如图 5-6 所示。

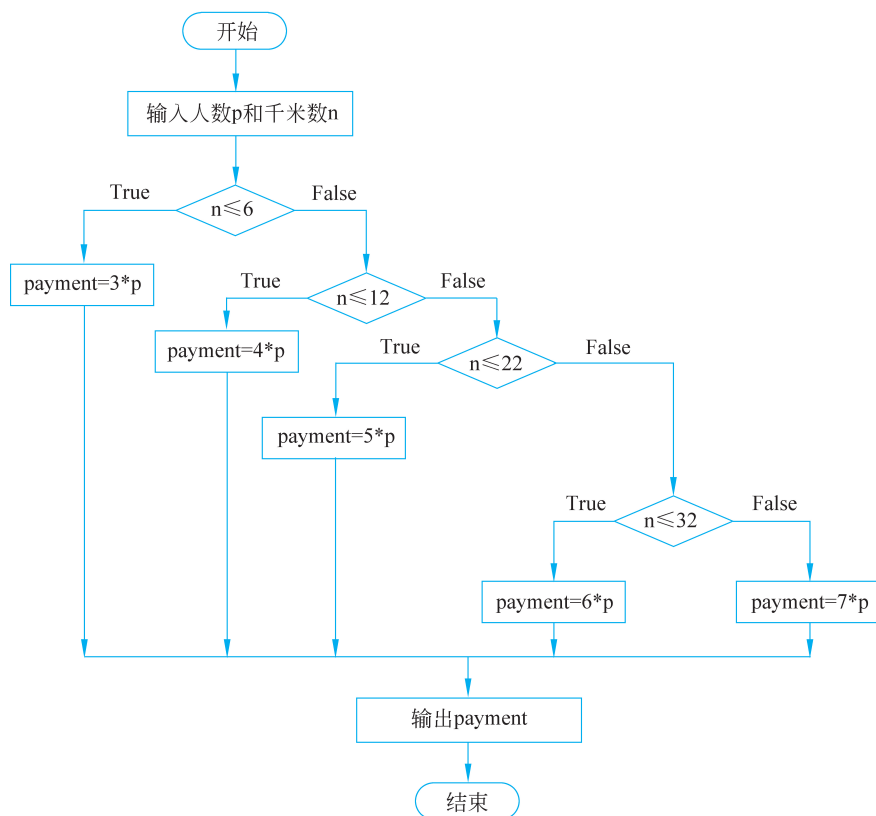


图 5-6 多分支结构费用计算

参考代码如下：

```

1. p=int(input("请输入人数"))
2. n=int(input("请输入千米数"))
3. if n<=6:
4.     payment=3*p
5. elif n<=12:
6.     payment=4*p
7. elif n<=22:
8.     payment=5*p
9. elif n<=32:
10.    payment=6*p
11. else:
12.    payment=7*p
13. print("您应支付的金额为",payment)
  
```

注意代码第 13 行 print() 是没有缩进的,表示它不属于最后一个 else 中的代码块,而是

与 if、elif、else 这些语句是一个级别,也就是说,无论是上面哪个分支结构,运行后都会进入第 13 行的 print() 进行输出。

多重分类情况也可以用分支嵌套来实现,分支嵌套是指在一个分支结构内部又存在一个分支结构。内部的分支结构可以嵌套在 if 子句下,也可以嵌套在 else 子句下。例如:

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

或

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

分支嵌套中的各级缩进必须正确,同一级缩进要一致。例 5.4 用分支嵌套实现如下:

```
p=int(input("请输入人数"))
n=int(input("请输入公里数"))
if n<=6:
    payment=3*p
else:
    if n<=12:
        payment=4*p
    else:
        if n<=22:
            payment=5*p
        else:
            if n<=32:
                payment=6*p
            else:
                payment=7*p
print("您应支付的金额为",payment)
```

尽管很多情况下多分支结构与分支嵌套可以解决相同的问题,但二者在结构、逻辑和使用场景上还是有所区别的。

- 结构上: 多分支结构是线性的,每个条件是独立的;而分支嵌套是层级的,条件是嵌套的。
- 逻辑上: 多分支结构通常用于根据单一变量的不同值来选择不同的执行路径;分支嵌套则用于在已经满足某个条件的基础上进一步细化条件。
- 使用场景: 多分支结构适用于条件较为简单且并列的情况;分支嵌套适用于条件较为复杂,需要在不同层级上进行判断的情况。



## 5.3 循环结构

程序代码重复做某件事的现象称为循环。在使用循环时必须弄清楚：重复做的工作是什么(循环体)；重复的条件(循环条件)是什么。

根据循环执行次数是否确定，循环可以分为确定次数循环和非确定次数循环。确定次数循环指循环条件中的循环次数有明确的定义，这类循环在 Python 中称为遍历循环，其中，循环次数由遍历结构中的元素个数确定。遍历循环采用 for...in... 结构实现。非确定次数循环中循环体的执行次数不确定，通过循环条件判断是否继续执行循环体，这类循环采用 while 语句实现。

### 5.3.1 遍历循环

遍历循环的语法结构如下：

```
for <循环变量> in <遍历结构>:  
    <语句块>
```

遍历循环是一种确定次数的循环，其循环次数由遍历结构中的元素个数控制。遍历结构可以是数值序列、字符串、列表、文件等。语句块(循环体)是用来循环执行的语句。每次循环时，从遍历结构中取一个元素放到循环变量，并执行一次语句块。完整遍历结构中的所有元素后，循环结束(图 5-7)。

例如：

```
for i in range(1,10,2):  
    print(i)
```

运行结果：

```
1  
3  
5  
7  
9
```

遍历结构 range(1,10,2)产生的数据序列为[1,3,5,7,9]，所以共循环 5 次，循环变量 i 每次从序列中取一个元素，所以输出为 1,3,5,7,9。

例如：

```
for i in ['a','b','c','d',1,2]:  
    print("Hello")
```

运行结果：

```
Hello  
Hello  
Hello
```

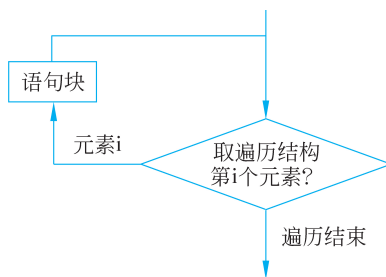


图 5-7 遍历循环结构

```
Hello
Hello
Hello
```

本例中遍历结构 `['a','b','c','d',1,2]` 为包含 6 个元素的列表,所以执行 6 次循环体,共输出 6 次 Hello。本例中,我们并不关心循环变量的值,只是想要循环 6 次,这种情况也可以用下划线“\_”表示循环变量。上述代码也可以写为

```
for _ in ['a','b','c','d',1,2]:
    print("Hello")
```

**例 5.5** 用 for 循环求 1~100 的累加和。

```
s=0
for i in range(1,101):
    s=s+i
print("和为",s)
```

本例中,s 用来保存累加和,其初始值为 0。range(1,101)产生一个 1~100 的数据序列,这个序列正好是要求和的加数。每次循环时,用上一次的累加和 s 加上当前的循环变量值。所有数字累加后,输出 s 值。第 4 行 print() 语句与 for 对齐,表示循环结束后,输出 s 值,如果 print() 与 s=s+i 对齐,则表示 print() 是循环体的一部分,每循环一次,便执行一次 print() 语句。

**例 5.6** 例 4.6 中用顺序结构绘制了矩形,其实只需要绘制出矩形的一个长和宽,另外的长和宽是相同的操作,因此可以利用循环绘制矩形。利用循环绘制 660×440、填充颜色为 #DE2910 的矩形,结果如图 5-8 所示。(eg5\_7\_drawRectangle.py)

参考代码:

```
import turtle
t=turtle.Turtle()
t.speed(1)                                #设置速度
t.penup()
t.goto(-330, 220)                         #设置起始位置
t.pendown()
t.color("#DE2910")                        #设置颜色
t.begin_fill()
for _ in range(2):
    t.forward(660)
    t.right(90)
    t.forward(440)
    t.right(90)
t.end_fill()
t.hideturtle()
turtle.done()
```

turtle 默认绘图起始位置位于窗口中心,先利用 penup() 提起画笔,然后利用 t.goto(-330, 220) 将画笔移到左上角(-330,220)的位置,再利用 pendown() 放下画笔,准备绘图。代码第 9~13 行利用 for 循环实现了矩形的绘制。

**例 5.7** 利用循环绘制五角星。

思路分析:

五角星有多种绘制方法,假设以图 5-9 所示的思路绘制五角星。先绘制五角星的一条



边,然后右转  $144^\circ$  准备绘制另一条边,这样的操作重复 5 次,即可完成五角星的绘制。(eg5\_8\_drawStar.py)



图 5-8 带填充的矩形

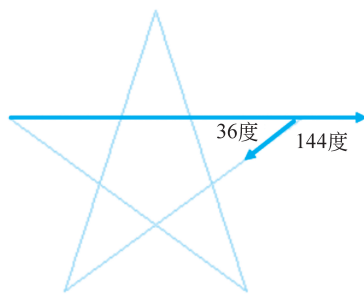


图 5-9 五角星绘制原理

参考代码如下:

```
import turtle
t=turtle.Turtle()
t.color("#FFDE00")
t.penup()
t.goto(-50, -50)           #设置起始位置
t.pendown()
t.begin_fill()
for _ in range(5):
    t.forward(100)
    t.right(144)
t.end_fill()
t.hideturtle()
turtle.done()
```

### 5.3.2 条件循环

很多循环无法事先知道循环次数,如猜数字游戏,如果猜错了,则继续猜,直到猜对为止。因为无法预知几次可以猜对,所以无法使用前面的遍历循环。这种情况下,需要使用条件循环来实现。

条件循环的基本语法格式如下:

```
while <条件表达式>:
    <语句块>
```

当条件满足时,执行循环体中的语句块,条件不满足时结束循环。while 循环一般用于不确定循环次数的情况,对于循环次数确定的内容,也可以用 while 循环。例如例 5.6 实现 1~100 的累加和,用 while 循环结构的实现如下:

```
i=1                #循环变量的初始值
s=0                #累加的和初始值
while i<=100:
    s=s+i
    i+=1           #循环变量变化
print("累加和为",s)
```

循环条件是  $i \leq 100$ , 第 1 行  $i=1$  为循环变量的初始值, 第 5 行  $i+=1$  为循环变量的变化, 每循环一次, 循环变量加 1。在使用 while 循环时, 一定要注意循环变量的初始值, 以及循环变量的变化。如果循环初始值不合适, 则会导致无法进入循环。如果在循环过程中循环变量没有变化, 则会进入死循环, 进而造成程序崩溃。

## 5.4 break 语句和 continue 语句

在处理问题时, 有时需要提前结束循环, Python 中用 break 语句和 continue 语句可以实现此功能。break 语句用来结束所在循环, 程序从循环后的代码处继续执行。continue 语句用来结束当次循环, 不再执行循环体后面尚未执行的语句。

例如:

```
for i in range(1,10):
    if i==5:
        break
    print(i,end=" ")
```

程序运行结果为:

```
1 2 3 4
```

当  $i=5$  时, 执行 break 语句, 结束 for 循环, 因此输出为 1 2 3 4。

将上述代码中的 break 改为 continue, 再看程序运行结果:

```
for i in range(1,10):
    if i==5:
        continue
    print(i,end=" ")
```

程序运行结果为:

```
1 2 3 4 6 7 8 9
```

程序运行结果为 1 2 3 4 6 7 8 9, 没有输出 5, 这是因为在  $i=5$  时运行了 continue 语句, 结束了当前循环不进行输出, 接着进行下一轮循环。

**例 5.8** “合抱之木, 生于毫末; 九层之台, 起于累土; 千里之行, 始于足下”。一张厚度为 0.1 毫米的足够大的纸, 对折多少次以后才能达到珠穆朗玛峰的高度? (eg5\_10\_foldPaper.py)

思路分析:

用 fThinkness 表示纸张的厚度, 初始值为 0.1 毫米, 即 0.0001m, 每对折一次, fThinkness 变成原来的 2 倍, 即  $fThinkness * = 2$ , 珠穆朗玛峰的高度是 8848.86 米, 当  $fThinkness > 8848.86$  时, 结束循环, 但折叠多少次满足条件尚不能确定, 所以本例适合用 while 循环完成(图 5-10)。