第3章

程序控制结构

CHAPTER 3

本章学习目标：

- 理解条件表达式的值与 True 或 False 的等价关系；
- 熟练掌握选择结构；
- 熟练掌握循环结构；
- 熟练掌握循环结构中 break 和 continue 语句的用法；
- 理解带有 else 子句的循环结构的执行过程。

3.1　结构化程序设计

结构化程序设计（Structured Programming）是一种经典的程序设计方法，旨在通过简洁、清晰的程序结构提升代码的可读性、可维护性和可扩展性。该方法强调使用简单且明确的控制结构来组织程序逻辑，避免复杂的跳转和冗余代码，从而提高程序的质量和稳定性。

结构化程序设计的核心包括三大控制结构：顺序结构、选择结构和循环结构。这些控制结构构成了结构化程序的基础，帮助开发者以更清晰、有效的方式组织和控制程序流程。

在结构化程序设计中，程序流程图（FlowChart）和 N-S 图（Nassi-Shneiderman Diagram）是最常用的工具，用于描述程序的控制流和逻辑结构。

程序流程图也称为程序框图，是一种图形化工具，用于表示程序的执行流程或算法步骤。它通过一系列标准符号，清晰地描述了程序从开始到结束的执行过程，包括各个步骤的顺序、决策条件和循环过程，帮助开发者、设计人员及其他相关人员快速理解程序的逻辑结构。

图 3-1 展示了几种常见的流程图标准符号。

图 3-1　常见的流程图标准符号

（1）起止框（椭圆形）：用于标示流程的开始和结束。通常，框内标注“开始”或“结束”以明确流程的起始和终止节点。

（2）输入输出框（平行四边形）：表示程序中进行数据输入或输出操作的步骤。框内通常标注“输入……”或“打印/显示……”等内容，用以指示数据的输入或结果的输出。

（3）判断框（菱形）：用于判断给定条件，根据条件是否满足来决定流程的后续路径。判断框一般有一个入口和两个出口，当条件成立时，出口标注为“是”或“Y”；条件不成立时，出口标注为“否”或“N”。

（4）处理框（矩形）：表示执行某项操作，如计算或数据处理。框内通常填写具体的操作内容，例如赋值操作或其他处理步骤。

（5）流程线（箭头）：指示流程的流向，表示各步骤之间的执行顺序。

（6）连接点（圆形）：用于连接多个流程图部分或分支点，简化复杂的流程图。连接点能够有效地连接不同部分，避免流程线交叉或过长，使流程图更加清晰易懂。

（7）注释框：注释框并非流程图的必要组成部分，它不涉及流程和操作的具体内容。其主要作用是对流程图中某些步骤或操作进行补充说明，以帮助阅读者更好地理解流程图的含义和目的。

程序流程图的主要作用：

（1）**可视化**：通过图形化方式展示程序结构和执行流程，帮助阅读者快速理解程序的步骤和逻辑。它将复杂的程序或算法形象化，简化了抽象概念的表达，使得程序更易于

理解。

（2）**辅助设计**：在程序编码之前，设计人员可以先绘制流程图，明确各个步骤的顺序和相互关系。这有助于减少逻辑错误，尤其在团队协作中，流程图作为一种有效的沟通工具，能够确保开发人员、设计人员和测试人员之间对程序设计的理解一致。

（3）**文档化**：流程图作为程序文档的一部分，便于团队成员之间的协作与知识共享，同时为后期的维护工作提供清晰的参考资料，增强程序的可维护性。

（4）**调试与优化**：在开发或调试过程中，程序员可以通过流程图迅速定位问题，发现潜在的错误和性能瓶颈，从而更高效地进行程序优化。

程序流程图是程序设计与分析中的重要工具。通过图形化的展示，它在设计、沟通、文档化、调试和优化等各个环节中都发挥着关键作用。无论是在编写代码之前的设计阶段，还是在后期的维护与优化过程中，流程图都能够提供极大的帮助。

3.2 顺序结构

顺序结构（Sequential Structure）指的是程序中的代码按照顺序逐行执行的控制结构。换句话说，程序中的语句从上到下按顺序依次执行，且没有任何跳转或分支，直到程序执行完毕。

顺序结构是最基本且最简单的控制结构。

例 3-1：顺序结构程序示例。

在 IDLE 编辑器中，新建 3-1.py 文件，输入以下代码并运行。

```
1  print('Hello, Python!')  # 输出'Hello, Python!'
2  a = 5  # 定义变量 a 并赋值为 5
3  b = 10  # 定义变量 b 并赋值为 10
4  c = a + b  # 计算 a + b，并将结果赋值给变量 c
5  print('The sum of a and b is:', c)  # 输出计算结果
Out Hello, Python!
    The sum of a and b is: 15
```

顺序结构是 Python 程序的默认执行方式。在例 3-1 中，代码从第 1 行到第 5 行按顺序逐行执行，整个过程简洁明了，易于理解。

例 3-2：输入一个三位自然数，计算并输出百位、十位和个位数字。

在 IDLE 编辑器中，新建 3-2.py 文件，输入以下代码并运行。

解法一：

```
1  num = int(input())  # 获取用户输入的三位数字字符串，并将其转换为整数
2  a = num // 100  # 提取百位数字
3  b = num // 10 % 10  # 提取十位数字
4  c = num % 10  # 提取个位数字
5  print(a, b, c)  # 输出各位数字
In 123
```

Out	1 2 3

解法二：

1	num = input()　# 获取用户输入的三位数字字符串
2	a, b, c = map(int, num)　# 将字符串的各个字符都转换为整数，并通过解包赋值给 a、b 和 c
3	print(a, b, c)
In	123
Out	1 2 3

小技巧：序列解包（Sequence Unpacking）是 Python 中的一种便捷操作，允许将一个序列（如列表、元组、字符串）中的多个元素直接分配给多个变量。通过这种方式，Python 能够自动拆分序列，并按照顺序将每个元素分别赋值给对应的变量。

3.3　选择结构

选择结构，又称为分支结构，是根据特定条件决定程序执行路径的一种控制结构。选择结构通常分为三种类型：单分支选择结构、二分支选择结构和多分支选择结构，这三种选择结构通过不同的条件判断来控制程序的流向。

在 Python 中，选择结构通常通过 if、elif 和 else 语句来实现。

3.3.1　单分支选择结构

语法格式如下：

```
if 条件表达式：  # 注意：冒号不能缺少，用于引导当条件成立时要执行的语句块
    <语句块>  # 当条件表达式为真时，执行该语句块（即条件执行体）
```

特别注意：在 Python 中，语句块指的是由一条或多条语句组成的、具有相同缩进级别的代码段。通常由控制结构（如 if、for、while 等）决定其执行范围。语句块中的所有语句作为一个整体，要么全部执行，要么都不执行。

如图 3-2 所示，当条件表达式的值等价于 True 时，表示条件成立；当条件表达式的值等价于 False 时，表示条件不成立。程序根据条件的成立与否，决定是否执行相应的语句块（即条件执行体）。

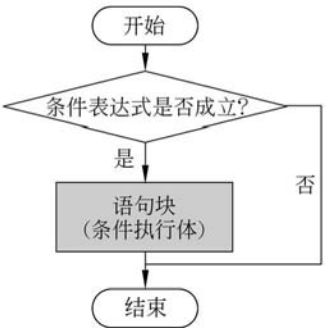

图 3-2　单分支选择结构

在编程中，我们常常结合使用关系运算符（如==、!=、>、>=、<、<=）和逻辑运算符（如 and、or、not）来构造条件表达式。

在 Python 中，以下对象会被视为等价于 False：0、0.0、0j、None、False、空字符串""、空列表[]、空元组()、空字典{}、空集合 set()、空 range 对象，以及其他使 bool()函数返回 False 的对象。除这些值外，其他对象都被视为等价于 True。

例 3-3：Python 中各类对象与 True 或 False 的等价关系。

```
1 >>> print(bool(0), bool(0.0), bool(0j), bool(None), bool(''))
2 Out False False False False False
3 >>> print(bool([]), bool(()), bool(set()), bool({}), bool(range(0)))
4 Out False False False False False
5 >>> print(bool(-2), bool(3.14), bool(4j), bool('OK'))
6 Out True True True True
7 >>> print(bool([1]), bool((2,)), bool({3}), bool({'Age': 28}), bool(range(4)))
8 Out True True True True True
```

例 3-4：输入两个以空格分隔的整数，并按升序输出。

在 IDLE 编辑器中，新建 3-4.py 文件，输入以下代码并运行。

```
1  num = input()  # 获取用户输入的内容，并将 input()函数返回的字符串赋值给变量 num
2  a, b = map(int, num.split())  # 分割字符串 num 并转换为整数，再将它们赋值给变量 a 和 b
3  if a > b:  # 判断 a 是否大于 b
4      a, b = b, a  # 如果 a > b，则交换 a 和 b 的值
5  print(a, b)  # 输出 a 和 b
In 5 3
Out 3 5
```

第 1 行：使用 input()函数获取用户从键盘输入的内容，默认将其作为字符串处理。用户需要输入两个整数，且用空格分隔。例如，用户输入“5 3”，那么输入的字符串会被存储在 num 变量中，形式为'5 3'。

第 2 行：num.split()方法会将输入的字符串 num 按照空格进行分割，返回一个列表。例如，'5 3'.split()会返回['5', '3']。接着，map(int, ...)函数会将列表中的每个字符串元素都转换为整数，因此'5'和'3'会分别被转换为整数 5 和 3。然后，a, b = ...将转换后的整数分别赋值给变量 a 和 b。在这个例子中，a 被赋值为 5，b 被赋值为 3。

第 3 行：这行代码是一个条件判断，用来检查 a 是否大于 b。

第 4 行：如果 a > b 为 True，则交换 a 和 b 的值，使得较小的数赋值给 a，较大的数赋值给 b，从而保证两个数按从小到大的顺序排列。如果 a > b 为 False，则交换操作不会执行，a 和 b 的值保持不变。

第 5 行：最后，输出 a 和 b 的值，即按从小到大的顺序打印这两个整数。如果用户输入的是“5 3”，经过交换后，输出将是“3 5”。

在 Python 中，如果 if 语句的条件执行体（即语句块）非常简洁，可以将其直接写在冒号后面，这种写法被称为单行 if 语句或单行条件语句。

例 3-5：单行条件语句示例。

```
1 >>> a, b = 5, 3
2 >>> if a > b: a, b = b, a  # 单行 if 语句，适用于条件执行体非常简洁的情况
3 ...  ↵
4 >>> print(a, b)
```

```
5  Out  3 5
6  >>>  if 5 > 3: print(5); print(3)    # 单行 if 语句中可以使用分号分隔多条语句
7  ...  ↵
8  Out  5
        3
```

单行 if 语句适用于条件执行体非常简洁的情况。当条件执行体较为复杂或包含多条语句时，建议使用常规的结构化语句块形式（即通过冒号、换行和缩进的多行写法），以提高代码的可读性。

此外，类似的还有单行循环语句，例如单行 for 循环和单行 while 循环等，这些语句也应根据实际情况灵活选择。

例 3-6：单行循环语句示例。

```
1  >>>  # pass 是 Python 中的空语句，其作用是占位，不执行任何操作
2  >>>  for i in range(5): pass    # 循环体为空，pass 语句用于占位，使 for 循环的语法结构完整
3  ...  ↵
4  >>>  n = 0
5  >>>  while n < 5: n += 1
6  ...  ↵
```

3.3.2　二分支选择结构

语法格式如下：

```
if 条件表达式：  # 注意：冒号不能缺少，用于引导当条件成立时要执行的语句块
    <语句块 1>   # 当条件为真时，执行语句块 1
else:
    <语句块 2>   # 当条件为假时，执行语句块 2
```

如图 3-3 所示，当条件表达式的值等价于 True 时，表示条件成立，程序将执行语句块 1；当条件表达式的值等价于 False 时，表示条件不成立，程序将执行语句块 2。

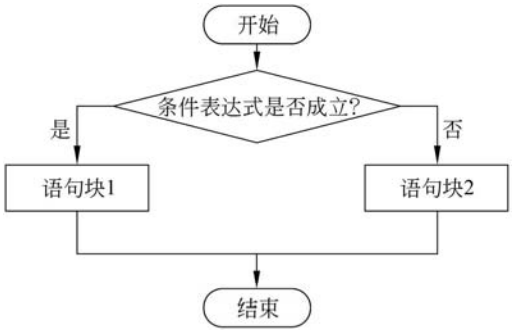

图 3-3　二分支选择结构

例 3-7：编写程序，根据输入的年龄判断是否为成年人（年龄大于或等于 18 岁）。

在 IDLE 编辑器中，新建 3-7-1.py 文件，输入以下代码并运行。

```
1   age = int(input())   # 获取用户输入的年龄，并将其转换为整数后赋值给变量 age
2   if age >= 18:   # 判断年龄是否大于或等于 18
3       print('成年人')   # 条件为真时执行 if 语句块中的代码
4   else:
5       print('未成年人')   # 条件为假时执行 else 语句块中的代码
In  25
Out 成年人
```

例 3-8：输入两个以空格分隔的整数，判断第一个数是否为第二个数的因子。

在 IDLE 编辑器中，新建 3-8.py 文件，输入以下代码并运行。

```
1   num = input()   # 获取用户输入的内容，并将 input()函数返回的字符串赋值给变量 num
2   a, b = map(int, num.split())   # 分割字符串 num 并转换为整数，再将它们赋值给变量 a 和 b
3   if b % a == 0:   # 判断 a 是否是 b 的因子，使用%运算符求余并判断余数是否为 0
4       print('{}是{}的因子'.format(a, b))   # 如果 b 除以 a 的余数为 0，说明 a 是 b 的因子
5   else:
6       print('{}不是{}的因子'.format(a, b))   # 否则，a 不是 b 的因子
In  12 36
Out 12 是 36 的因子
```

此外，Python 还提供了三元运算符（条件表达式），用于简洁地表示二分支的 if 语句。其语法格式如下：

```
表达式 1 if 条件表达式 else 表达式 2
```

当条件表达式的值等价于 True 时，整个表达式的值为表达式 1 的值；当条件表达式的值等价于 False 时，整个表达式的值为表达式 2 的值。与传统的 if-else 语句相比，条件表达式在语法上更加简洁紧凑。

以例 3-7 为例，在 IDLE 编辑器中，新建 3-7-2.py 文件，输入以下代码并运行。

```
1   age = int(input())   # 获取用户输入的年龄，并将其转换为整数后赋值给变量 age
2   res = '成年人' if age >= 18 else '未成年人'   # 根据年龄是否大于或等于 18，返回对应的字符串
3   print(res)   # 输出判断结果
```

这种写法简洁且易于理解，适用于处理简单的条件判断。

例 3-9：计算 ISBN-13 条形码的校验码。首先，计算前 12 位数字的加权和：奇数位乘以 1，偶数位乘以 3。然后，将加权和除以 10，得到余数。如果余数为 0，校验码为 0；否则，校验码为 10 减去余数。

在 IDLE 编辑器中，新建 3-9.py 文件，输入以下代码并运行。

```
1   isbn = '978703069977'   # ISBN 前 12 位
2   total = 0   # 初始化加权和为 0
3   for i, digit in enumerate(isbn):   # 计算加权和
4       if i % 2 == 0:   # 奇数位
5           total += int(digit)
```

```
6       else:   # 偶数位
7           total += int(digit) * 3
8   remainder = total % 10
9   check_digit = 0 if remainder == 0 else 10 - remainder   # 计算校验码
10  print('校验码：{}'.format(check_digit))
Out 校验码：0
```

3.3.3　多分支选择结构

语法格式如下：

```
if 条件表达式 1:  # 注意：冒号不能缺少，用于引导当条件 1 成立时要执行的语句块
    <语句块 1>  # 当条件 1 为真时，执行该语句块
elif 条件表达式 2:
    <语句块 2>  # 当条件 1 为假且条件 2 为真时，执行该语句块
…
elif 条件表达式 n:
    <语句块 n>  # 当条件 1~n-1 都为假且条件 n 为真时，执行该语句块
else:
    <语句块 n+1>  # 如果以上条件都为假时，执行该语句块
```

如图 3-4 所示，在 if-elif-else 结构中，只有第一个符合条件的语句块会被执行，后续的 elif 或 else 语句都将被跳过。换句话说，一旦某个条件表达式成立，程序会执行相应的语句块并跳过其后的条件判断，即使其他条件表达式为真，它们的语句块也不会被执行。

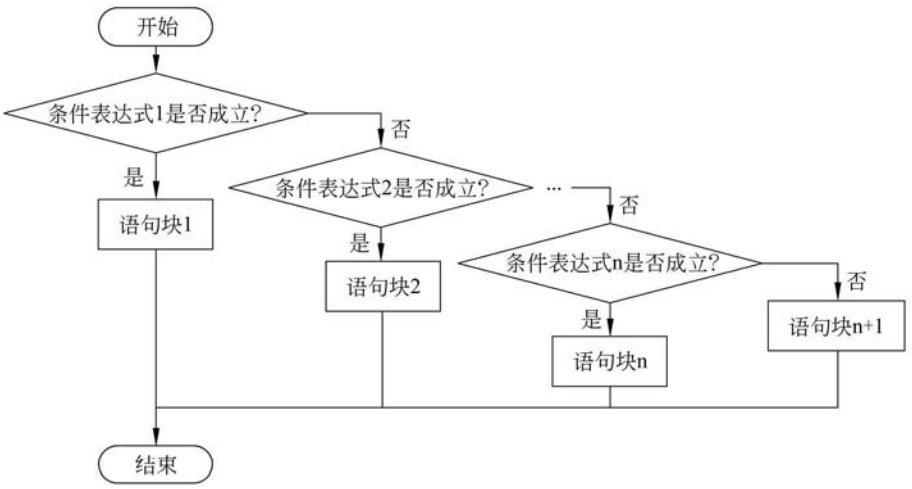

图 3-4　多分支选择结构

例 3-10：根据输入的分数输出对应的成绩等级。

在 IDLE 编辑器中，新建 3-10.py 文件，输入以下代码并运行。

```
1   score = int(input())   # 获取用户输入的成绩，并将其转换为整数后赋值给变量 score
2   if score < 0 or score > 100:   # 判断成绩是否在合理范围内
```

```
    res = '输入的成绩不在合理范围内'
elif score >= 90:
    res = '优秀'
elif score >= 80:
    res = '良好'
elif score >= 70:
    res = '中等'
elif score >= 60:
    res = '及格'
else:
    res = '不及格'
print(res)  # 输出成绩对应的等级
In  95
Out 优秀
```

归纳起来，Python 的选择结构可以通过 if 语句、任意数量的 elif 语句和最多一个的 else 语句来实现。if 语句是选择结构的基础，不可或缺；elif 语句是 else if 的缩写，属于可选部分，可以包含 0 个或多个 elif 分支；else 语句同样是可选的，用于处理所有条件都不满足时的默认情况。

3.3.4 选择结构的嵌套

选择结构的嵌套是指在一个选择语句（如 if-elif-else 语句）内部再次使用选择语句。通过这种方式，可以实现多层次的条件判断，从而处理更加复杂的业务逻辑。

其语法格式如下：

```
if 条件表达式 1:  # 注意：冒号不能缺少，用于引导当条件 1 成立时要执行的语句块
    <语句块 1>  # 当条件 1 为真时，执行该语句块
    if 条件表达式 1-1:
        <语句块 1-1>  # 当条件 1 和条件 1-1 都为真时，执行该语句块
    elif 条件表达式 1-2:
        <语句块 1-2>  # 当条件 1 为真，且条件 1-1 为假、条件 1-2 为真时，执行该语句块
    else:
        <语句块 1-3>  # 当条件 1 为真，但条件 1-1 和条件 1-2 均为假时，执行该语句块
elif 条件表达式 2:
    <语句块 2>  # 当条件 1 为假，且条件 2 为真时，执行该语句块
else:
    <语句块 3>  # 当条件 1 和条件 2 均为假时，执行该语句块
```

例 3-11：输入整数，判断是否能被 2 或 3 整除。

在 IDLE 编辑器中，新建 3-11.py 文件，输入以下代码并运行。

```
num = int(input())  # 获取用户输入，并将其转换为整数后赋值给变量 num
if num % 2 == 0:  # 判断 num 是否能被 2 整除
    if num % 3 == 0:  # 判断 num 是否能被 3 整除
```

```
4           print(num, '能被 2 和 3 整除。')
5       else:
6           print(num, '能被 2 整除，但不能被 3 整除。')
7   else:
8       if num % 3 == 0:    # 判断 num 是否能被 3 整除
9           print(num, '能被 3 整除，但不能被 2 整除。')
10      else:
11          print(num, '不能被 2 或 3 整除。')
In  6
Out 6 能被 2 和 3 整除。
```

嵌套选择结构需要严格缩进，以确保不同层级的代码块正确对齐。此外，应尽量避免过深的嵌套层次，以免代码变得难以理解和维护。因此，在逻辑设计时，可以通过优化业务逻辑或拆分复杂逻辑等方式，减少嵌套深度，从而提高代码的可读性和可维护性。

3.4　循环结构

Python 中常用的循环结构包括 for 循环和 while 循环，用于执行重复性的任务。for 循环通常用于遍历序列（如列表、元组、字符串）或其他可迭代对象，而 while 循环则适用于在满足某个条件时反复执行代码块。

3.4.1　for 循环

for 循环用于遍历列表、元组、字符串等可迭代对象中的每个元素，或者配合 range() 函数执行指定次数的重复任务，如图 3-5 所示。

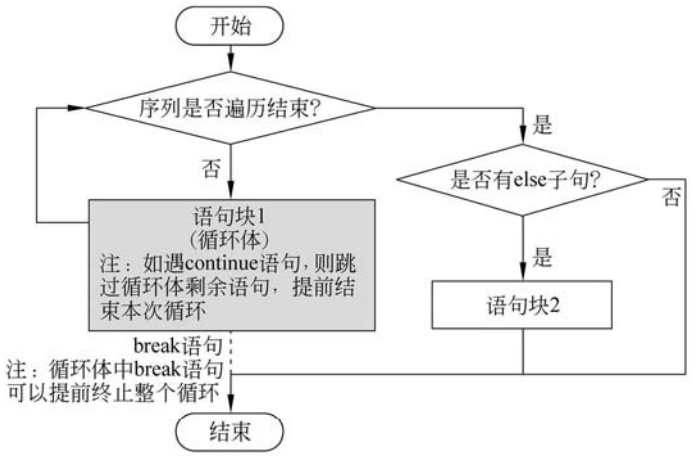

图 3-5　for 循环结构

其语法格式如下：

```
for item in iterable:
```

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

其中，item 是每次迭代时当前元素的值，iterable 是待遍历的序列，可以是列表、元组、字符串、range 对象等。

for 循环会遍历序列中的每个元素，直到遍历完成，循环正常结束后会执行 else 子句中的语句块 2。如果循环是由于碰到 break 语句而提前终止，则 else 子句不会执行。

例 3-12：for 循环示例。

1	>>>	fruits = ['apple', 'banana', 'cherry']
2	>>>	for fruit in fruits: # 遍历列表中的元素
3	...	print(fruit)
4	...	↵
5	Out	apple banana cherry
6	>>>	for i in range(5): # range(5)生成一个从 0 到 4 的整数序列，常用于指定循环次数
7	...	print(i, end=' ')
8	...	↵
9	Out	0 1 2 3 4

例 3-13：大学四年学习与成长的历程。

在 IDLE 编辑器中，新建 3-13.py 文件，输入以下代码并运行。

1	base = 1 # 初始值设为 1
2	for i in range(365 * 4): # range(365*4)生成从 0 到 1459 的整数序列，表示 4 年的天数
3	base = base * (1 + 0.001) # 每天进步一点点（0.1%）
4	print(base)
Out	4.302819417401616

在追求目标的过程中，切勿忽视那些看似微不足道的细节和小步伐。任何伟大的成就都源于日常点滴努力的积累。无论是在学习、工作，还是在人生的其他目标上，日积月累的努力和坚持才是突破和进步的关键。正如古人所言：“滴水穿石”，“不积跬步，无以至千里”。

3.4.2 while 循环

语法格式如下：

```
    while 条件表达式:
        <语句块 1>
    else:
        <语句块 2>
```

如图 3-6 所示，while 循环会在条件表达式的值等价于 True 时不断重复执行语句块 1，直到条件表达式的值等价于 False，此时循环正常结束，并执行 else 子句中的语句块 2。如

果循环是由于碰到 break 语句而提前终止，则 else 子句则不会执行。

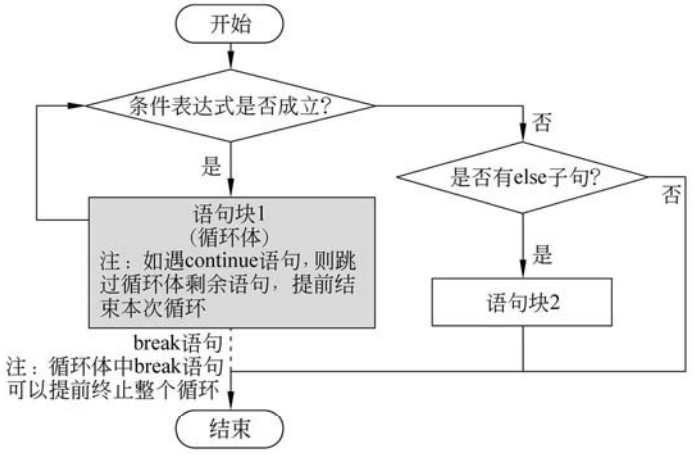

图 3-6　while 循环结构

例 3-14：计算 1～100 的总和。

在 IDLE 编辑器中，新建 3-14.py 文件，输入以下代码并运行。

```
1   total = 0   # 初始化总和 total 为 0
2   i = 1   # 从 1 开始遍历
3   while i <= 100:   # 当 i 小于或等于 100 时，执行循环体
4       total += i   # 将当前的 i 加到 total 上，等价于 total = total + i
5       i += 1   # 将 i 增加 1，计算下一项
6   print(total)   # 输出累加结果
Out 5050
```

例 3-15：猜数字游戏。

在 IDLE 编辑器中，新建 3-15.py 文件，输入以下代码并运行。

```
1   import random   # 导入 random 模块，用于生成随机数
2
3   target = random.randint(10, 99)   # 生成一个随机的两位数作为目标数字
4   count = 1   # 初始化猜测次数
5   guess = int(input('请输入两位整数：'))   # 获取用户输入并转换为整数后赋值给变量 guess
6   while guess != target:   # 循环直到猜对为止
7       if guess < target:   # 猜测小了
8           print('你输入的数字小了，再试试！')
9       else:   # 猜测大了
10          print('你输入的数字大了，再试试！')
11      guess = int(input('请输入两位整数：'))   # 获取下一次猜测
12      count += 1   # 猜测次数加 1
13  print('恭喜你，猜对了！你一共猜了', count, '次。')   # 猜对后输出结果
In  请输入两位整数：54
Out 你输入的数字大了，再试试！
```

```
In  请输入两位整数：31
Out 你输入的数字小了，再试试！
In  请输入两位整数：42
Out 恭喜你，猜对了！你一共猜了 3 次。
```

3.4.3 continue 和 break 语句

在 Python 中，for 循环和 while 循环均可使用 continue 和 break 语句，以便更灵活地控制循环的执行流程。

1．continue 语句

continue 语句用于跳过当前循环体中剩余的代码，马上进入下一次循环的条件判断。它不会终止整个循环，而是让程序跳过本次循环后续的代码，然后进入下一次循环。通常在满足特定条件时使用，以跳过不需要处理的情况。

例 3-16：打印 1 到 10 之间的所有奇数。

在 IDLE 编辑器中，新建 3-16.py 文件，输入以下代码并运行。

```
1  for i in range(1, 11):   # 遍历从 1 到 10 的所有整数
2      if i % 2 == 0:   # 如果 i 是偶数
3          continue   # 跳过后续的代码即 print 语句，马上进入下一次循环
4      print(i, end=' ')
Out 1 3 5 7 9
```

在例 3-16 中，当 i 为偶数时，continue 语句会被执行，从而跳过循环体中后续的 print(i) 语句，提前结束本次循环并直接进入下一次循环。因此，程序最终只会打印出奇数。

2．break 语句

break 语句用于提前终止其所在的循环，立即跳出该循环体。一旦 break 语句被执行，不管循环条件是否满足，当前循环将立即结束，程序将继续执行循环之后的代码。

例 3-17：输入一个整数，判断其是否为素数（素数是大于 1 且仅能被 1 和自身整除的自然数）。

在 IDLE 编辑器中，新建 3-17.py 文件，输入以下代码并运行。

```
1  n = int(input())   # 获取用户输入，并将其转换为整数后赋值给变量 n
2  res = True   # 初始化结果变量 res，不妨先假设 n 是素数
3  if n <= 1:   # 判断 n 是否小于或等于 1
4      res = False   # 若是，则它不是素数
5  else:
6      # 只需检查到 √n ，因为一个大于 √n 的因子必然存在一个小于 √n 的因子与之对应
7      for i in range(2, int(n**0.5) + 1):   # i 从 2 开始遍历到 √n
8          if n % i == 0:
9              res = False   # 如果 n 能被 i 整除，则 n 不是素数
```

```
10                  break   # 找到因子后可以提前结束循环，以提高效率
11 if res:   # 根据 res 的值，输出判断结果
12     print('{}是素数'.format(n))
13 else:
14     print('{}不是素数'.format(n))
In  197
Out 197 是素数
```

在 Python 中，无限循环（也称为死循环）是指一个永远不会自动终止的循环，除非通过外部操作（如手动停止）或循环内部的控制语句（如 break 语句）来打断。无限循环通常用于需要持续运行的程序，例如服务程序、事件监听、定时轮询等。

例 3-18：使用 while 或 for 创建无限循环。

在 IDLE 编辑器中，新建 3-18.py 文件，输入以下代码并运行。

示例一：

```
1 while True:
2     print('这是一个无限循环')
```

示例二：

```
1 for _ in iter(int, 1):    # iter(int, 1)会返回一个无限的迭代器
2     print('这也是一个无限循环')
```

示例三：

```
1 i = 0   # 初始化计数器 i
2 while i <= 100:   # 当 i 小于或等于 100 时，重复执行循环体
3     if i == 10:
4         continue   # 当 i 等于 10 时，跳过本次循环后续的代码，马上进入下一次循环
5     i += 1   # 计数器自增 1
```

例 3-19：计算阶乘并演示如何使用 break 语句跳出无限循环。

在 IDLE 编辑器中，新建 3-19.py 文件，输入以下代码并运行。

```
1  n = int(input())   # 获取用户输入，并将其转换为整数后赋值给变量 n
2  fac, i = 1, 1   # 阶乘和计数器的初始值均设为 1
3  while True:   # 循环条件始终为真，因此循环会无限执行下去
4      fac *= i   # 累乘，等价于 fac = fac * i
5      i += 1
6      if i > n:   # 当 i 大于 n 时，执行 break 语句终止循环
7          break
8  print(fac)
In  5
Out 120
```

例 3-20：编写程序，持续接收用户输入并回显，输入 exit 时自动终止。

在 IDLE 编辑器中，新建 3-20.py 文件，输入以下代码并运行。

```
while True:
    txt = input()   # 获取用户输入的字符串
    if txt == 'exit':   # 当用户输入 exit 时
        break   # 结束无限循环
    else:
        print(txt)   # 显示用户输入的内容
```

如果死循环没有终止条件，它可能导致程序持续运行，占用大量 CPU 资源，从而影响系统性能。在某些情况下，死循环还可能导致程序崩溃或引发内存泄漏。因此，在编写代码时，应避免创建没有终止条件的死循环，或确保有适当的机制可以中断它。

3.4.4 else 语句

for 循环和 while 循环后可以附加一个 else 子句。只有当循环正常结束（即没有因为碰到 break 语句提前终止）时，else 子句中的语句块才会执行。

注意：else 子句中的语句块并不属于循环体，而是与循环的控制结构密切相关。

例 3-21：输出 200 以内的最大素数。

在 IDLE 编辑器中，新建 3-21.py 文件，输入以下代码并运行。

```
for n in range(200, 1, -1):   # 逐个检查[200, 2]范围内的每个数是否为素数
    for i in range(2, int(n**0.5) + 1):   # 检查 n 是否能被[2, √n ]范围内的任何数整除
        if n % i == 0:   # 如果 n 能被 i 整除，说明 n 不是素数
            break   # 提前终止内层循环，此时 else 子句中的语句块不会执行
    else:   # 如果内层循环是正常结束的，说明没有找到 n 的任何因子，即 n 是素数
        print(n)   # 输出最大素数
        break   # 找到最大素数后，通过 break 语句提前终止外层循环
Out 199
```

在例 3-21 中，外层 for 循环从 200 开始，向下遍历到 2，逐个检查每个整数是否为素数。内层 for 循环用于检查当前数字 n 是否存在除 1 和 n 以外的因子。如果找到因子（即 n % i 等于 0），说明 n 不是素数，内层循环会通过 break 语句提前结束，直接进入外层循环的下一次循环。如果内层循环没有被 break 语句终止，意味着没有找到 n 的任何因子，说明 n 是素数，内层循环会正常结束，接着执行内层 for 循环的 else 子句中的语句块，输出 n 并通过 break 语句提前结束外层循环。

3.5 异常处理

异常（Exception）是指程序执行过程中发生的错误，通常会造成程序中断或产生不正确的结果。Python 提供了强大的异常处理机制，允许开发者在错误发生时将其捕获并妥善处理，从而确保程序的稳定性和持续运行。

3.5.1　异常类型

Python 中的异常种类繁多，常见的异常类型及其功能描述如表 3-1 所示。

表 3-1　常见的异常类型及其功能描述

异常类型	功能描述
SyntaxError	语法错误，表示代码的语法不符合 Python 语言的规则，通常在程序的解析阶段就会被检测到，比如写错关键字、遗漏括号等
IndentationError	缩进错误，通常在代码块的缩进不正确时会抛出该异常
TypeError	类型错误，表示操作数的类型与操作不匹配，例如在对字符串和数字进行加法运算时，就会抛出该异常
ValueError	值错误，通常发生在传递给函数的参数值不符合预期的情况，例如将无法转换为整数的字符串传递给 int()函数时，就会抛出该异常
IndexError	索引错误，在访问序列（如列表、元组、字符串等）时，使用的索引超出了序列的有效范围，就会抛出该异常
KeyError	键错误，在访问字典中不存在的键时会抛出该异常
AttributeError	属性错误，在尝试访问对象不存在的属性或方法时会抛出该异常
ZeroDivisionError	除零错误，在尝试进行除法操作且除数为零时会抛出该异常
FileNotFoundError	文件未找到错误，在尝试打开一个不存在的文件时会抛出该异常
ImportError	导入错误，当模块或模块中的某个函数、类无法导入时会抛出该异常
OverflowError	溢出错误，通常发生在进行算术运算时，运算结果超出了数据类型所能表示的范围
MemoryError	内存错误，当程序尝试使用的内存超过了系统可用的内存资源时会抛出该异常
RecursionError	递归错误，通常发生在递归调用的深度超过了递归深度限制，导致程序无法继续执行下去。Python 中默认递归深度限制是 1000 次，超过这个深度会抛出该异常
Exception	BaseException 的子类，能够捕获大多数常见的错误和异常，但不包括 SystemExit、KeyboardInterrupt 和 GeneratorExit 等系统级别的异常
BaseException	所有内建异常类的基类，理论上可以用于捕获所有类型的异常。然而，通常不建议直接捕获 BaseException，除非有特殊的需求

这些异常类型有助于识别和处理程序中的各种错误，从而提高代码的鲁棒性和可维护性。有关异常的示例，请参见本书 1.4.3 节。

3.5.2　异常处理

在程序运行过程中，若出现异常，不仅需要捕获该异常，还需根据异常的具体原因判断其类型，并制定相应的处理方案，以确保程序能够继续稳定运行。Python 提供了 try-except 语句、else 子句和 finally 子句来处理异常，这使得异常处理更加灵活和完善。

其语法格式如下：

```
try:
    <语句块 1>   # 放置可能会引发异常的代码
except 异常类型 1:
    <语句块 2>   # 当发生类型 1 异常时，执行该语句块
```

```
…
except 异常类型 n：
    <语句块 n+1>  # 当发生类型 n 异常时，执行该语句块
else：
    <语句块 n+2>  # 如果 try 块中没有发生任何异常，则执行该语句块
finally：
    <语句块 n+3>  # 无论 try 块是否发生异常，都会执行该语句块
```

try 块的作用是包裹可能会引发异常的代码，若 try 块中引发异常，则 Python 会按照 except 子句的排列顺序逐个检查，一旦找到与该异常类型匹配的 except 子句，就会执行该子句中的语句块，并且不会再去检查后续的 except 子句，同时也跳过 else 子句；若 try 块中没有引发任何异常，程序会跳过所有的 except 子句，直接执行 else 子句中的语句块。

无论 try 块中是否发生异常，也无论异常是否已被 except 子句捕获，finally 子句中的语句块都会被执行。它通常用于执行一些清理操作，如关闭文件、释放资源等。

注意：try 语句必须与 except 子句或者 finally 子句配合使用。

例 3-22：输入整数索引，返回字符串“ABCDEFGHIJKLMNOPQRSTUVWXYZ”中对应的字符。

在 IDLE 编辑器中，新建 3-22.py 文件，输入以下代码并运行。

```
1  chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
2  try:
3      n = int(input('请输入字符的索引：'))  # 获取用户输入，并将其转换为整数后赋值给变量 n
4      print(f'{chars[n]}')  # 使用 f 字符串输出指定索引的字符
5  except ValueError:
6      print('值错误：索引必须是合法的整数')
7  except IndexError:
8      print('索引错误：索引超出有效范围[0, 25]')
9  except Exception as e:
10     print('其他错误：', e)  # 捕获所有其他异常并打印错误信息
11 else:
12     print('try 语句块没有发生异常')  # 如果没有发生异常，执行该语句块
13 finally:
14     print('程序执行完毕')  # 无论是否发生异常，都会执行该语句块
```

请运行上述程序，分别输入以下值：5、3.14 和 100，并观察程序的运行结果及其差异。

3.6 经典案例解析

例 3-23：打印九九乘法表。

在 IDLE 编辑器中，新建 3-23.py 文件，输入以下代码并运行。

```
1  for i in range(1, 10):  # 遍历 1 到 9 的数字作为行数
2      for j in range(1, i + 1):  # 遍历 1 到 i 的数字作为列数
```

```
            # 打印乘法公式，使用 str.format()格式化输出，'{:<3d}'确保乘积左对齐且占 3 个字符宽度
            print('{} * {} = {:<3d}'.format(i, j, i * j), end='')
        print()  # 每输出一行后换行，end 参数默认为换行符\n
```

例 3-24：输出所有各位数字不同的三位数。

基础解法：在 IDLE 编辑器中，新建 3-24-1.py 文件，输入以下代码并运行。

```
digits = range(10)  # 生成 0～9 的整数序列
for i in digits:  # i 为百位数，从 0 开始遍历到 9
    if i == 0:
        continue  # 百位数不能是 0，跳过此情况，通过 continue 语句提前结束本次循环
    for j in digits:  # j 为十位数，从 0 开始遍历到 9
        if j == i:
            continue  # 十位数不能与百位数相同，跳过此情况
        for k in digits:  # k 为个位数，从 0 开始遍历到 9
            if k in (i, j):
                continue  # 个位数不能与百位数或十位数相同，跳过此情况
            print(i, j, k, sep='')  # 输出符合条件的三位数
```

高级解法：在 IDLE 编辑器中，新建 3-24-2.py 文件，输入以下代码并运行。

```
for num in range(100, 1000):  # 遍历所有三位数
    # 将数字转换为集合，检查各位数字是否均不相同
    set_num = set(str(num))  # 先将数字转换为字符串，再将字符串转换为集合
    if len(set_num) == 3:  # 集合会自动去重，若去重后集合长度为 3，则表示数字各位均不相同
        print(num)  # 输出符合条件的三位数
```

例 3-25：从键盘输入一个字符串 s，对其进行加密。加密规则如下：对于英文大小写字母，每个字母向后循环移动 6 位（例如：A 变成 G，B 变成 H，…，Z 变成 F；a 变成 g，b 变成 h，…，z 变成 f）；其他字符保持不变。

测试样例：

```
请输入字符串进行加密：I love China.
加密后的字符串：O rubk Inotg.
```

注：每个字符在计算机中都对应着一个特定的 ASCII 码值，可以使用 **ord**()函数来查询该值。例如，ord('0') = 48，ord('1') = 49，ord('A') = 65，ord('B') = 66，ord('a') = 97，ord('b') = 98，以此类推。反过来，可以使用 **chr**()函数将一个 ASCII 码值转换为与之对应的字符。例如，chr(48)返回'0'，chr(65)返回'A'，chr(97)返回'a'，等等。

在 IDLE 编辑器中，新建 3-25.py 文件，输入以下代码并运行。

```
s = input('请输入字符串进行加密：')  # 获取用户输入的字符串
encrypted_s = ''  # 存储加密后的字符串，初始值设为空字符串
for char in s:  # 遍历输入字符串 s 中的各个字符
    if 'A' <= char <= 'Z':  # 若 char 是大写字母，向后循环移动 6 位
        encrypted_s += chr((ord(char) - ord('A') + 6) % 26 + ord('A'))
```

```
6       elif 'a' <= char <= 'z':   # 若 char 是小写字母，向后循环移动 6 位
7           encrypted_s += chr((ord(char) - ord('a') + 6) % 26 + ord('a'))
8       else:
9           encrypted_s += char   # 其他字符保持不变，直接添加到加密后的字符串中
10  print('加密后的字符串：{}'.format(encrypted_s))
In  请输入字符串进行加密：I love China.
Out 加密后的字符串：O rubk Inotg.
```

例 3-26：利用莱布尼茨公式 $\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}+\cdots$ 来计算 π，并将结果精确到小数点后七位。

延伸阅读：祖冲之（429 年—500 年），字文远，我国伟大的数学家和天文学家。他运用精妙的割圆术，将圆周率的计算精确到小数点后 7 位，这一成就堪称非凡，直至千年之后，阿拉伯数学家阿尔·卡西才打破这一纪录。

在 IDLE 编辑器中，新建 3-26.py 文件，输入以下代码并运行。

```
1   pi = 0   # 存储 π/4 的近似值
2   i = 0   # 迭代次数
3   while True:   # 迭代计算莱布尼茨公式的每一项，直到精度满足要求
4       term = (-1)**i / (2 * i + 1)   # 计算莱布尼茨公式的当前项
5       pi += term   # 累加当前项到 pi 中
6       if abs(term) < 10**(-8):   # 如果当前项的绝对值小于给定精度 10^-8，则停止迭代
7           break   # 通过 break 语句跳出循环
8       i += 1   # 迭代次数加 1，继续下一次迭代
9   pi *= 4   # 计算最终的π值
10  print('{:.7f}'.format(pi))   # 输出结果，保留小数点后 7 位
Out 3.1415927
```

梅钦公式（Machin-like formula）是一类用于高效计算圆周率 π 的数学公式，基于反正切函数的级数展开。最著名的梅钦公式由英国数学家约翰·梅钦（John Machin）于 1706 年提出，其经典形式为：

$$\pi=16\cdot\arctan\left(\frac{1}{5}\right)-4\cdot\arctan\left(\frac{1}{239}\right)$$

反正切函数 arctan(x)的泰勒级数展开式为：

$$\arctan(x)=x-\frac{x^3}{3}+\frac{x^5}{5}-\frac{x^7}{7}+\cdots$$

通过梅钦公式，可以逐项计算反正切函数的级数，从而高效地逼近圆周率π。与莱布尼茨公式相比，梅钦公式具有更快的收敛速度，因此它成为了计算圆周率的更高效、实用的方法。时至今日，梅钦公式及其变种仍然是圆周率计算的重要工具之一。

本章习题

3.1　以下选项不属于程序流程图基本元素的是（　　）。

A．循环框　　B．连接点　　C．判断框　　D．起止框

3.2　关于结构化程序设计所要求的基本结构，以下选项中描述错误的是（　　）。

A．顺序结构　　B．选择（分支）结构

C．goto 跳转　　D．重复（循环）结构

3.3　以下关于程序控制结构描述错误的是（　　）。

A．分支结构包括单分支结构和二分支结构

B．二分支结构组合形成多分支结构

C．程序由三种基本结构组成

D．Python 里，能用分支结构写出循环的算法

3.4　关于 Python 的分支结构，以下选项中描述错误的是（　　）。

A．分支结构使用 if 保留字

B．Python 中 if-else 语句用来形成二分支结构

C．Python 中 if-elif-else 语句描述多分支结构

D．分支结构可以向已经执行过的语句部分跳转

E．单分支结构是用 if 保留字判断满足一个条件，就执行相应的处理代码

3.5　以下关于循环结构的描述，错误的是（　　）（多选题）。

A．遍历循环使用 for <循环变量> in <循环结构>语句，其中循环结构不能是文件

B．使用 range()函数可以指定 for 循环的次数

C．for i in range(5)表示循环 5 次，i 的值是从 0 到 4

D．用字符串做循环结构的时候，循环的次数是字符串的长度

E．遍历循环的循环次数由遍历结构中的元素个数来体现

F．非确定次数的循环的次数是根据条件判断来决定的

G．非确定次数的循环用 while 语句来实现，确定次数的循环用 for 语句来实现

H．遍历循环对循环的次数是不确定的

3.6　以下关于分支和循环结构的描述，错误的是（　　）。

A．Python 在分支和循环语句里使用例如 x <= y <= z 的表达式是合法的

B．分支结构中的代码块是用冒号来标记的

C．while 循环如果设计不小心会出现死循环

D．二分支结构的<表达式 1> if <条件> else <表达式 2>形式，适合用来控制程序分支

3.7　若有 ls = [1, 2, 3, 4, 5, 6]，以下关于循环结构的描述，错误的是（　　）。

A．表达式 for i in range(len(ls))的循环次数跟 for i in ls 的循环次数是一样的

B．表达式 for i in range(len(ls))的循环次数跟 for i in range(0, len(ls))的循环次数是一样的

C．表达式 for i in range(len(ls))的循环次数跟 for i in range(1, len(ls) + 1)的循环次数是一样的