第八章 - 流程控制

张建章

阿里巴巴商学院

杭州师范大学

2024-09

流程控制是指在程序设计中，通过特定的语句和结构来控制程序执行的顺序和逻辑流向。

在商业数据处理领域，流程控制至关重要，因为它决定了数据处理的顺序、条件判断和循环操作，从而确保数据处理过程的准确性和效率。

例如，在处理客户订单时，使用条件语句可以根据订单状态采取不同的处理措施，而循环结构则可用于遍历大量数据记录进行批量处理。通过合理运用流程控制结构，能够构建出高效、可靠的数据处理流程，满足商业应用的需求。

Python 中的流程控制包括：顺序、选择、循环。

条件语句是程序设计中的基本控制结构，用于根据特定条件的真值判断，决定程序执行不同的代码块。在 Python 中，主要使用 `if`、`elif` 和 `else` 语句来实现条件判断。其基本语法如下：

```
if condition1:
    # 当 condition1 为 True 时执行的代码块
elif condition2:
    # 当 condition1 为 False 且 condition2 为 True 时执行的代码块
else:
    # 当上述条件均为 False 时执行的代码块
```

Python 使用缩进来表示代码块的范围。在条件语句中，缩进的代码块仅在条件为 `True` 时执行。

例如，以下代码根据输入的分数输出相应的成绩等级：

```
score = 85

if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
    grade = 'D'
else:
    grade = 'F'

print(f"Your grade is: {grade}")
```

在此示例中，程序根据`score`的值，依次判断各个条件，输出相应的成绩等级。这种条件判断结构在数据处理、决策分析等领域广泛应用。

缩进

在 Python 编程中，代码块的定义依赖于缩进，而非其他编程语言中常见的花括号或关键字。缩进的使用不仅影响代码的可读性，更是 Python 语法的核心部分。根据 Python 官方的 PEP 8 风格指南，建议每一级缩进使用四个空格。

以下示例展示了如何在 Python 中使用缩进来定义代码块：

```
if number % 2 == 0:
    print(f"{number} 是偶数。")
else:
    print(f"{number} 是奇数。")
```

在上述代码中，`if` 和 `else` 语句后的代码块通过缩进来表示其从属关系。如果缩进不一致，Python 解释器将抛出 `IndentationError`，提示缩进错误。

单选语句

单选语句（即 `if` 语句）用于根据特定条件的真值判断，决定是否执行某段代码。其基本语法如下：

```
if condition:
    # 当 condition 为 True 时执行的代码块
```

其中，`condition` 是一个布尔表达式。当 `condition` 为 `True` 时，缩进的代码块将被执行；否则，代码块将被跳过。

例如，以下代码根据输入的数字判断其正负性：

```
number = int(input(" 请输入一个整数: "))

if number > 0:
    print(" 该数字是正数。")
```

双选语句

双选语句（即 `if-else` 语句）用于根据条件的真值判断，决定程序执行不同的代码块。其基本语法如下：

```
if condition:
    # 当 condition 为 True 时执行的代码块
else:
    # 当 condition 为 False 时执行的代码块
```

例如，以下代码根据输入的年龄判断是否为成年人：

```
age = int(input(" 请输入年龄: "))

if age >= 18:
    print(" 您是成年人。")
else:
    print(" 您未成年。")
```

多选语句

多选语句（即 `if-elif-else` 语句）用于根据多个条件的真值判断，决定程序执行的代码块。其基本语法如下：

```
if condition1:
    # 当 condition1 为 True 时执行的代码块
elif condition2:
    # 当 condition1 为 False 且 condition2 为 True 时执行的代码块
elif condition3:
    # 当前面的条件均为 False 且 condition3 为 True 时执行的代码块
else:
    # 当上述所有条件均为 False 时执行的代码块
```

其中，`condition1`、`condition2`、`condition3` 等为布尔表达式。程序从上至下依次判断各条件，执行第一个为 `True` 的条件对应的代码块；如果所有条件均为 `False`，则执行 `else` 下的代码块。

循环结构是编程语言中的基本控制结构之一，用于重复执行特定代码块，直到满足指定的条件。在 Python 中，主要有两种循环结构：`for` 循环和 `while` 循环。`for` 循环用于遍历序列（如列表、元组、字符串等）或其他可迭代对象，而 `while` 循环则在给定条件为真时反复执行代码块。

在商业数据处理中，循环结构具有重要作用。它们用于自动化重复性任务，如批量处理数据记录、迭代计算统计指标、遍历数据集以查找特定模式或异常等。通过使用循环结构，可以提高数据处理的效率和准确性，减少人工操作的错误率。

`for` 循环用于遍历序列（如列表、元组、字符串等）或其他可迭代对象。其基本语法如下：

```
for item in iterable:
    # 执行的代码块
```

其中，`item` 是循环变量，在每次迭代中获取 `iterable` 中的下一个元素。`iterable` 是一个可迭代对象，如列表、元组、字符串、集合、字典或生成器（range、zip、enumerate）。

以下示例展示了如何使用 `for` 循环遍历列表中的元素：

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

`for` 循环依次将每个元素赋值给变量 `fruit`，并在循环体内打印该变量的值。

`while` 循环用于在指定条件为真时，反复执行代码块。基本语法：

```
while condition:
    # 执行的代码块
```

其中，`condition` 是一个布尔表达式。当 `condition` 为 `True` 时，循环体内的代码将被执行；当 `condition` 为 `False` 时，循环终止，程序继续执行后续代码。

以下示例展示了如何使用 `while` 循环打印 1 到 5 的数字：

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

需要**注意**的是，确保循环条件最终会变为 `False`，以避免出现无限循环。在上例中，通过在循环体内递增 `i`，保证了循环的正常结束。

break 语句和 continue 语句

在 Python 的**循环结构**中，`break` 和 `continue` 语句用于控制循环的执行流程。`break` 语句用于立即终止当前循环，跳出循环体，继续执行后续代码；`continue` 语句则用于跳过当前迭代，直接开始下一次循环。

1. `break` 语句的用法

`break` 语句通常用于在满足特定条件时退出循环。

```
# for 循环中的 break 示例
for number in range(1, 11):
    if number == 5:
        break
    print(number)
```

在此示例中，循环遍历数字 1 到 10。当 `number` 等于 5 时，`break` 语句被触发，循环立即终止，后续数字不再打印。

```python
# while 循环中的 break 示例
i = 1
while i <= 10:
    if i == 5:
        break
    print(i)
    i += 1
```

在此示例中，`while`循环不断增加变量`i`的值。当`i`等于 5 时，`break`语句被触发，循环立即终止。

2. `continue` 语句的用法

`continue` 语句用于跳过当前迭代的剩余代码，直接进入下一次循环迭代。

```
# for 循环中的 continue 示例
for number in range(1, 6):
    if number == 3:
        continue
    print(number)
```

在此示例中，循环遍历数字 1 到 5。当 `number` 等于 3 时，`continue` 语句被触发，当前迭代剩余代码被跳过，数字 3 未被打印，循环继续进行。

```python
# while 循环中的 continue 示例
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    print(i)
```

在此示例中，`while` 循环将变量 `i` 的值从 1 增加到 5。当 `i` 等于 3 时，`continue` 语句被触发，跳过当次打印操作，直接进入下一次循环。数字 3 未被打印。

`break` 和 `continue` 语句在控制循环流程中极为有用，能够灵活地管理循环终止与跳过的条件，但需谨慎使用，以避免产生难以调试的逻辑错误，如，无限循环。

3. 使用 while True 和 break 实现特定的循环控制

使用 while True 与 break 语句相结合，可以实现特定的循环控制。while True 创建一个无限循环，而 break 语句用于在满足特定条件时终止该循环。这种结构在需要持续运行某个过程，直到满足特定条件时尤为有用。

示例：用户输入验证

以下示例展示了如何使用 while True 和 break 语句实现用户输入验证，确保用户输入有效的整数：

```
while True:
    user_input = input(" 请输入一个整数: ")
    if user_input.isdigit():
        number = int(user_input)
        print(f" 您输入的整数是: {number}")
        break
    else:
        print(" 输入无效，请输入一个整数。")
```

代码解析：

- `while True`：创建一个无限循环，持续提示用户输入。
- `input()`函数：获取用户输入，并将其存储在`user_input`变量中。
- `if user_input.isdigit():`：检查用户输入是否为数字字符串。
- **如果是数字字符串：**
  - 将其转换为整数类型，并存储在`number`变量中。
  - 打印用户输入的整数。
  - 使用`break`语句终止循环。
- **如果不是数字字符串：**
  - 提示用户输入无效，要求重新输入。

通过这种方式，程序能够持续提示用户输入，直到获得有效的整数输入为止。

1. 嵌套 for 循环

多重 `for` 循环（即嵌套 `for` 循环）用于在一个 `for` 循环内部再嵌套一个或多个 `for` 循环，以遍历多维数据结构或生成复杂的迭代模式。这种结构在处理二维数组、矩阵运算或生成特定模式时尤为常见。

基本语法：

```
for outer_element in outer_sequence:
    for inner_element in inner_sequence:
        # 执行的代码块
```

在上述结构中，外层 `for` 循环遍历 `outer_sequence` 中的每个元素。对于外层循环的每次迭代，内层 `for` 循环都会遍历 `inner_sequence` 中的所有元素。这种嵌套关系可以扩展到多层次，但应注意层次过多可能导致代码复杂性增加。

示例：生成乘法表

以下示例展示了如何使用多重 `for` 循环生成一个 9×9 乘法表：

```
# 使用双重 for 循环打印 9×9 乘法表
for i in range(1, 10):  # 外层循环，控制乘法表的行数，从 1 到 9
    for j in range(1, i + 1):  # 内层循环，控制每行中列的输出范围
        print(f"{j}x{i}={i * j}", end="\t")  # 打印当前的乘法表达式，并用制表符隔开
    print()  # 每行结束后换行，进入下一行
```

2. 嵌套条件语句

嵌套条件语句是指在一个 `if` 或 `else` 块中包含另一个 `if` 语句。这在需要根据多个条件进行判断时非常有用。

```
x = 10
y = 5

if x > 0:
    if y > 0:
        print("x 和 y 都是正数")
    else:
        print("x 是正数，但 y 不是正数")
else:
    print("x 不是正数")
```

3. 条件语句与循环的嵌套

在循环内部使用条件语句，或在条件语句内部使用循环，是实现复杂逻辑的常见方式。

```
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num % 2 == 0:
        print(f"{num} 是偶数")
    else:
        print(f"{num} 是奇数")
```

在上述示例中，`for` 循环遍历列表中的每个数字，并使用 `if` 语句判断其奇偶性。

4. 嵌套循环与条件语句

在嵌套循环中使用条件语句，可以实现更复杂的逻辑控制。

```
for i in range(3):
    for j in range(3):
        if i == j:
            print(f"i 和 j 都是 {i}")
        else:
            print(f"i = {i}, j = {j}")
```

上述代码在 `i` 等于 `j` 时输出特定信息，否则输出 `i` 和 `j` 的值。

通过合理使用嵌套控制结构，可以编写出功能强大且灵活的程序，以满足复杂的业务需求。

`assert`、`pass`、`exec`和`eval`是四个重要的语句或函数，分别用于不同的场景。这些语句和函数在流程控制中扮演辅助或特殊用途角色。通过`assert`验证流程条件、`pass`填充流程结构、`exec`和`eval`实现动态代码执行，均可以提升代码的灵活性和适应性，使流程控制更加丰富和动态。

1. `assert`语句

`assert`用于在程序中插入调试断言。当条件为`False`时，程序会引发`AssertionError`异常。这在测试和调试时非常有用。

```
x = 10
assert x > 0, "x should be positive"
```

在上述代码中，如果`x`不大于 0，程序将抛出`AssertionError`，并显示消息"x should be positive"。

2. `pass` 语句

`pass` 是一个空操作，占位符语句。在需要语法上需要语句但不执行任何操作的地方使用。

```
for item in range(5):
    if item % 2 == 0:
        pass  # 占位符，无操作，程序继续执行
    else:
        print(item)
```

上例中，使用 `pass` 作为占位符，当 `item` 为偶数时，程序不进行任何操作。

3. exec 函数

exec 用于动态执行储存在字符串或文件中的 Python 代码。它可以执行更复杂的 Python 代码。

```
code = """
for i in range(5):
print(i)
"""
exec(code)
```

上述代码将动态执行字符串中的代码，输出 0 到 4。

4. `eval` 函数

`eval` 用于计算存储在字符串中的简单表达式，并返回结果。与 `exec` 不同，`eval` 只能处理单个表达式，不能执行复杂的代码结构。

```
expression = "3 * 4 + 5"
result = eval(expression)
print(result)  # 输出 17
```

在此示例中，`eval` 计算字符串中的表达式，并返回结果 17。

使用 `exec` 和 `eval` 时应谨慎，尤其是在处理不受信任的输入时，因为它们可能带来安全风险。

THE END