4 Control Flow

Control flow determines the order in which statements are executed in a program. In Python, control flow is primarily managed through conditional statements, loops, and control flow altering statements. These constructs allow a program to make decisions, iterate over data, and manage the sequence of execution dynamically.

Conditional statements such as if, if-else, and if-elif-else enable decision-making based on conditions that evaluate to True or False. Loops, including for and while, allow repetitive execution of code, making them useful for iterating over collections or executing tasks until a condition is met. Python also provides unique control structures, such as using else with loops and the match statement for pattern matching.

Additionally, control flow altering statements such as break, continue, and pass modify the behavior of loops and conditions, providing more flexibility. When combined with nested control structures, these tools form the foundation of complex decision-making and iterative processes in Python programs.

Topics Covered in This Chapter:

- The if Statement
- The Ternary Operator
- The match Statement
- The for Loop
- Range-Based Looping
- The while Loop
- Control Flow Altering Statements
- Nested Control Structures

This chapter provides a comprehensive guide to Python's control flow mechanisms, explaining how to use them effectively to build structured, logical, and efficient programs.

Conditional Statements

Control flow in Python is largely determined by conditional statements, which allow a program to make decisions based on specific conditions. These conditions are typically expressions that evaluate to True or False, utilizing Boolean logic. Conditional statements enable dynamic execution, where different code paths can be followed based on varying inputs or circumstances.

Python uses if, if-else, and if-elif-else statements to implement conditional logic. These structures help execute different blocks of code depending on the evaluation of a condition.

The if Statement

The if statement is the simplest form of a conditional statement in Python. It executes a block of code only if the given condition evaluates to True. If the condition is False, the code inside the if block is skipped.

Syntax

```
if condition:

    # Code to execute when condition is True
```

Example

```
x = 10

if x > 5:

    print("x is greater than 5")
```

Output:

```
x is greater than 5
```

If the condition inside if evaluates to False, the indented block is ignored.

Boolean Logic in if Statements

Conditions used in if statements often involve comparison operators (==, !=, >, <, >=, <=) or logical operators (and, or, not). Expressions that evaluate to True will execute the block inside the if statement.

Example Using Boolean Expressions

```
age = 20

if age >= 18:

    print("You are eligible to vote.")
```

Output:

```
You are eligible to vote.
```

Python considers certain values as False in a Boolean context, including 0, None, False, empty strings (""), empty lists ([]), and empty dictionaries ({}). Any other value is considered True.

Example with Truthy and Falsy Values

```
name = ""

if name:

    print("Name is provided.")  # This will not execute because an
empty string is falsy
```

Using if statements effectively allows programs to make decisions dynamically, forming the foundation of control flow in Python programs.

The Ternary Operator

The ternary operator in Python provides a concise way to write conditional expressions in a single line. It follows the format:

value_if_true if condition else value_if_false

Example:

```
age = 17

status = "Adult" if age >= 18 else "Minor"

print(status)
```

Output:

```
Minor
```

The ternary operator simplifies conditional assignments, making code more compact and readable.

Short-Circuiting in if Statements in Python

Short-circuiting is an optimization technique in Python where logical operators (and, or) stop evaluating as soon as the result is determined. This behavior is particularly useful inside if statements to improve efficiency and avoid unnecessary computations or errors.

Short-Circuiting with or

When using or, Python evaluates expressions from left to right and stops at the **first truthy value**. If the first value is truthy, the rest of the conditions are ignored.

Example:

```
x = 10

if x > 5 or some_function():

    print("Condition met")
```

Here, x > 5 is True, so some_function() is **never called**, preventing potential errors or unnecessary processing.

Short-Circuiting with and

When using and, Python evaluates expressions from left to right and stops at the **first falsy value**. If the first value is falsy, the rest of the conditions are ignored.

Example:

```
x = 2

if x > 5 and some_function():

    print("Condition met")
```

Since x > 5 is False, Python **never calls** some_function(), saving execution time.

Why is Short-Circuiting Useful?

1. **Prevents Errors** – Avoids executing functions that might cause exceptions if not needed.
2. **Improves Performance** – Skips unnecessary computations.
3. **Enhances Readability** – Allows for cleaner conditional checks.

Example where short-circuiting avoids an error:

```
value = None

if value and value > 10:

    print("Value is greater than 10")
```

Since value is None, the and condition stops before evaluating value > 10, preventing a TypeError.

The match Statement

Python 3.10 introduced the match statement, which allows for pattern matching similar to switch-case statements in other languages. It provides a more structured way to handle multiple conditions.

Syntax:

```
match variable:

    case pattern1:

        # Code for pattern1

    case pattern2:

        # Code for pattern2

    case _:

        # Default case (equivalent to 'else')
```

Example

```
code = 200

match code:

    case 200:

        print("OK")

    case 404:

        print("Not Found")

    case 500:

        print("Server Error")

    case _:

        print("Unknown Status")
```

The match statement is particularly useful when handling structured data or complex conditions, making it a powerful addition to Python's control flow mechanisms.

Looping Statements

Loops allow the execution of a block of code multiple times, enabling iteration over sequences or repeating a task until a condition is met. They are used when the number of iterations is unknown in advance or when automating repetitive tasks.

Python provides two primary looping constructs: for loops for iterating over sequences and while loops for executing code based on a condition.

The for Loop

The for loop is used for iterating over sequences such as lists, tuples, dictionaries, and strings. It simplifies repetitive tasks by automatically traversing elements in a collection.

Syntax:

```
for variable in sequence:

    # Code to execute on each iteration
```

Example: Iterating Over a List

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:

    print(fruit)
```

Output:

```
apple

banana

cherry
```

Using else with Loops *

In Python, loops can include an optional else block, which executes after the loop completes **only if the loop was not prematurely terminated by a break statement**. This behavior is unique to Python and can be useful in situations where you need to determine whether a loop finished naturally or was interrupted.

Explanation of the else Block in Loops

The else block in a loop is executed when the loop runs to completion without encountering a break. If the loop is terminated early with break, the else block is skipped. This feature applies to both for and while loops.

```
For example, let's consider searching for an element in a list:

numbers = [3, 7, 9, 12, 15]
```

```
for num in numbers:
    if num == 10:
        print("Found 10!")
        break
else:
    print("10 was not found in the list.")
```

Here, the loop checks each number in the list. If 10 is found, it prints a message and exits the loop using break. However, since 10 is not in the list, the loop runs to completion, triggering the else block, which informs us that 10 was not found.

The same concept applies to while loops. Consider a scenario where a program waits for user input:

```
attempts = 3

while attempts > 0:
    password = input("Enter password: ")
    if password == "secure123":
        print("Access granted.")
        break
    attempts -= 1
else:
    print("Too many failed attempts. Access denied.")
```

If the user enters the correct password within the allowed attempts, the loop breaks, and the else block does not execute. If the loop runs out of attempts without a correct password, the else block executes, displaying an "Access denied" message.

Demonstrating Usage After Loop Completion Without break

The else block is particularly useful when checking for conditions that should hold across an entire iteration. Consider checking whether all numbers in a list are positive:

```
numbers = [1, 5, 8, 12, 3]

for num in numbers:
    if num < 0:
        print("Negative number found!")
        break
else:
    print("All numbers are positive.")
```

Since there are no negative numbers in the list, the loop runs to completion, triggering the else block, which confirms that all numbers are positive. If a negative number were present, the break statement would prevent the else block from executing.

The else block in Python loops provides a way to distinguish between a loop that finishes normally and one that exits early due to break. It is especially useful for search operations, validation checks, and ensuring conditions hold across an entire loop. Understanding this feature allows for cleaner, more efficient loop structures that clearly convey intent in Python code.

Range-Based Looping

The range() function generates a sequence of numbers, commonly used for iterating a specific number of times.

Example:

```
for i in range(5):

    print(i)
```

Output:

```
0

1

2

3

4
```

The while Loop

The while loop executes a block of code as long as a given condition remains True. It is useful when the number of iterations is unknown beforehand.

Syntax:

```
while condition:

    # Code to execute repeatedly while condition is True
```

Example: Looping with a Condition

```
x = 0

while x < 5:

    print(x)

    x += 1
```

Output:

```
0

1

2
```

```
3

4
```

Infinite Loops

A while loop that never meets its exit condition results in an infinite loop. To prevent this, ensure the loop condition eventually evaluates to False or break out .

Example of an Infinite:

```
while True:

    print("This will run forever!")
```

To stop an infinite loop, use a break statement or ensure a condition eventually turns False.

Example Using break to Exit a Loop:

```
x = 0

while True:

    print(x)

    x += 1

    if x == 5:

        break
```

Output:

```
0

1

2

3

4
```

Loops are essential for automating repetitive tasks, processing data, and managing control flow efficiently in Python programs.

Control Flow Altering Statements in Python

In Python, control flow refers to the order in which statements are executed in a program. By default, statements run sequentially, following the logic of loops and conditions. However, in certain cases, it is necessary to modify this flow by either exiting a loop prematurely, skipping an iteration, or inserting a placeholder for future logic. Python provides three key statements to achieve this: break, continue, and pass.

The break Statement

The break statement is used to exit a loop immediately, regardless of the loop's natural termination condition. Normally, loops continue running until their condition evaluates to False, but break allows the program to interrupt this execution based on a specific condition.

For example, suppose we have a list of numbers, and we want to find the first even number, stopping the loop as soon as we find it:

```
numbers = [3, 7, 9, 4, 8, 10]


for num in numbers:

    if num % 2 == 0:

        print(f"First even number found: {num}")

        break
```

Here, the loop iterates through the list and stops once it encounters 4, the first even number, without checking the remaining elements.

The break statement is also useful inside while loops, particularly for loops that are meant to run indefinitely but need an exit condition:

```
while True:

    user_input = input("Enter a number (type 'exit' to stop): ")

    if user_input.lower() == "exit":

        break

    print(f"You entered: {user_input}")
```

This loop runs continuously, waiting for user input, but terminates immediately when the user types "exit".

The continue Statement

Unlike break, which stops a loop entirely, continue skips the current iteration and proceeds to the next one. This is useful when certain conditions require ignoring specific values without stopping the loop completely.

For instance, let's say we want to print only the odd numbers from a given list:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]


for num in numbers:

    if num % 2 == 0:

        continue  # Skip even numbers

    print(num)
```

Whenever an even number is encountered, continue skips the print(num) statement and moves directly to the next iteration. As a result, only odd numbers are displayed.

This approach is useful when filtering data or avoiding unnecessary operations in loops.

The pass Statement

Unlike break and continue, which actively alter the control flow of loops, pass does nothing. It serves as a placeholder where a statement is syntactically required but no action is needed at the moment.

Consider a function that has been defined but not yet implemented:

```
def future_function():

    pass  # Placeholder for future implementation
```

Without pass, an empty function body would cause a syntax error. pass allows the function to be defined while keeping the code valid.

Similarly, it can be used inside conditional statements when a block of code is required but will be written later:

```
number = 10


if number > 0:

    pass  # Logic will be added later

else:

    print("Number is non-positive.")
```

Even though the if condition is met, pass ensures that the code does not break due to an empty block.

Python's control flow altering statements help modify the natural flow of loops and conditions. The break statement allows for early termination, continue skips iterations based on conditions, and pass serves as a placeholder for incomplete code. These statements make Python code more flexible, readable, and efficient, allowing developers to handle various scenarios effectively.

Nested Control Structures

In programming, control structures like loops and conditional statements dictate the flow of execution. Often, it is necessary to use these structures within one another, creating **nested control structures**. Nesting occurs when an if statement is placed inside a loop, a loop is placed inside another loop, or an if statement is embedded within another if statement. While nesting is a powerful technique, excessive or poorly structured nesting can lead to code that is difficult to read and maintain.

Combining if Statements and Loops

A common use case for nested control structures is filtering data within loops. For example, consider a scenario where we need to find all even numbers greater than 10 from a given list:

```
numbers = [3, 12, 5, 18, 9, 21, 10, 6, 14]
```

```
for num in numbers:

    if num % 2 == 0:  # Check if the number is even

        if num > 10:   # Check if the number is greater than 10

            print(f"Valid number: {num}")
```

Here, the outer for loop iterates through the list, while the nested if statements apply additional filters. The first if ensures that only even numbers are considered, and the second if further restricts the output to numbers greater than 10.

Similarly, nested loops can be used when dealing with multi-dimensional data structures, such as lists of lists:

```
matrix = [

    [1, 2, 3],

    [4, 5, 6],

    [7, 8, 9]

]



for row in matrix:

    for num in row:

        if num % 2 == 0:

            print(f"Even number found: {num}")
```

In this example, the outer for loop iterates over rows in the matrix, while the inner loop goes through each number in the current row. The nested if statement checks for even numbers and prints them when found.

Explanation of Nesting and Best Practices for Readability

While nesting is a natural part of programming, deeply nested structures can quickly make code hard to read and debug. Here are some best practices to maintain clarity and improve readability:

Avoid excessive nesting – When multiple levels of if statements or loops are necessary, consider refactoring your code. For example, using the continue statement can sometimes eliminate unnecessary levels of nesting:

```
for num in numbers:

    if num % 2 != 0:

        continue  # Skip odd numbers

    if num > 10:

        print(f"Valid number: {num}")
```

Here, continue helps avoid an extra level of indentation by skipping numbers that do not meet the criteria, leading to cleaner code.

Use functions to encapsulate logic – If a block of nested code performs a distinct task, move it into a function. This improves modularity and reusability:

```
def is_valid_number(num):
    return num % 2 == 0 and num > 10


for num in numbers:
    if is_valid_number(num):
        print(f"Valid number: {num}")
```

This approach improves readability and makes the logic easier to test and modify.

Consider using list comprehensions – When working with collections, list comprehensions can simplify nested structures:

```
valid_numbers = [num for num in numbers if num % 2 == 0 and num > 10]
print(valid_numbers)
```

This eliminates explicit loops and conditionals, making the code more concise.

Use meaningful indentation and comments – Clearly separating different levels of logic and providing concise comments helps maintain readability in deeply nested structures.

Nested control structures are a fundamental concept in programming, allowing for complex decision-making and data processing. However, excessive nesting can make code harder to read and maintain. By following best practices—such as avoiding deep nesting, breaking logic into functions, using list comprehensions, and leveraging continue—you can write cleaner, more efficient Python programs while preserving readability and maintainability.

Summary

Control flow is the backbone of any Python program, dictating how statements execute based on conditions and loops. This chapter explored the fundamental mechanisms that allow programs to make decisions, repeat tasks, and modify execution dynamically.

Conditional statements such as if, if-else, and if-elif-else provide a way to execute different code paths depending on whether a condition evaluates to True or False. Boolean logic plays a crucial role in these decisions, using comparison and logical operators to control the flow of execution. The ternary operator offers a more concise way to express simple conditional assignments, while the match statement, introduced in Python 3.10, allows for structured pattern matching similar to switch-case statements in other languages.

Loops enable repetitive execution, with for loops iterating over sequences and while loops running based on conditions. Python also provides unique features such as using else with loops, which executes a block of code only if the loop completes without encountering a break. Range-based looping simplifies iteration over a predefined sequence of numbers, and handling infinite loops properly ensures programs do not get stuck in unintended endless execution.

To further refine control flow, Python provides the break, continue, and pass statements. The break statement allows loops to exit early when a specific condition is met, while continue skips the current iteration and moves to the next one. The pass statement serves as a placeholder, ensuring syntactic correctness without affecting execution.

Nested control structures allow for more complex decision-making by combining multiple loops and conditionals, though excessive nesting can lead to code that is difficult to read and maintain. Best practices, such as reducing unnecessary indentation, using helper functions, and leveraging list comprehensions, help keep nested logic clear and manageable.

By understanding and applying these control flow principles effectively, Python developers can write more structured, efficient, and readable code. This chapter laid the foundation for handling decision-making, iteration, and program execution in a way that enhances both functionality and maintainability.

Exercises

1. **Basic Conditional Statement**
   Write a program that asks the user for their age and prints whether they are a minor (under 18), an adult (18–64), or a senior citizen (65 and older). Use if, elif, and else statements.
2. **Ternary Operator**
   Rewrite the following conditional statement using the ternary operator:
3. score = int(input("Enter your test score: "))
4. if score >= 50:
5.   result = "Pass"
6. else:
7.   result = "Fail"
8. print(result)
9. **Using the match Statement**
   Write a function that takes a day of the week (as a string, e.g., "Monday") and returns whether it is a weekday or weekend using the match statement.
10. **Looping with for and range**
    Write a program that prints all even numbers between 1 and 20 using a for loop and the range() function.
11. **Finding the First Prime Number**
    Write a program that checks each number in a list to find the **first prime number** and prints it. Use a loop and break to exit when the first prime is found.
12. **While Loop with User Input**
    Create a program that repeatedly asks the user to enter a number. If the number is negative, print "Negative number detected, exiting..." and break the loop. Otherwise, continue prompting the user.

13. **Using continue in a Loop**
    Write a program that prints all numbers from 1 to 10 except multiples of 3. Use continue to skip the multiples.

14. **Nested Loops: Multiplication Table**
    Write a program that generates a multiplication table from 1 to 5. Use nested loops to iterate through rows and columns.

15. **Simulating a Simple ATM System**
    Write a program that simulates an ATM. It starts with a balance of $1000. The user can input **"withdraw"**, **"deposit"**, or **"exit"**. If they choose **"withdraw"**, prompt for an amount and deduct it from the balance (unless the amount exceeds the balance). If they choose **"deposit"**, prompt for an amount and add it to the balance. The program should continue running until the user enters "exit".

16. **Guess the Number Game**
    Write a program that randomly selects a number between 1 and 50 and asks the user to guess it. The program should:

    - Give hints whether the guess is too high or too low.
    - Allow up to **5 attempts** before revealing the correct number.
    - Exit if the user guesses correctly.