## Book Preview

This is a preview version containing only the first 2 chapters of the book.

**Full Book Details:**

- **Title:** Practical Python Programming
- **Subtitle:** Concepts, Codes and Solutions
- **Author:** Kindson Munonye
- **Edition:** 1st Edition
- **Publisher:** Alkademy Books

To access the complete book with all chapters, additional content, and full features, please purchase the full version.

*Thank you for your interest!*

# Contents

# CHAPTER 1

# CONDITIONAL STATEMENTS

Conditional statements in Python allow the program to make decisions and execute specific blocks of code based on whether certain conditions are met. Python uses the keywords if, elif, and else for this purpose.

Conditional statements are fundamental constructs in programming that allow a program to execute different blocks of code based on certain conditions. In Python, these statements enable developers to implement logic that can make decisions, control the flow of execution, and handle various scenarios dynamically. Understanding conditional statements is crucial for writing effective and efficient Python programs.

In this chapter, we will cover the various types of conditional statements in Python, their syntax, usage, and best practices. We will explore if, elif, and else statements, nested conditionals, logical operators, and advanced conditional constructs like the ternary operator. Practical examples will illustrate how to apply these concepts in real-world programming tasks.

## 1.1 Understanding Conditional Statements

Conditional statements evaluate expressions and execute code blocks based on whether the expressions are True or False. They are essential for controlling the flow of a program, enabling it to respond differently under varying conditions.

**Basic Structure:**

```
if condition:
    # Code block executed if condition is True
elif another_condition:
    # Code block executed if another_condition is True
else:
    # Code block executed if all above conditions are False
```

## 1.2 The if Statement

The if statement is the cornerstone of conditional logic in Python. It evaluates a condition and executes a block of code if the condition is True.

**Syntax:**

```
if condition:
    # Code to execute if condition is True
```

**Example:**

```
age = 20

if age >= 18:
    print("You are eligible to vote.")
```

Output:

You are eligible to vote.

## 1.3 The elif Statement

The elif (short for "else if") statement allows you to check multiple expressions for True and execute a block of code as soon as one of the conditions is met.

**Syntax:**

```
if condition1:
    # Code to execute if condition1 is True
elif condition2:
    # Code to execute if condition2 is True
```

**Example:**

```
score = 85

if score >= 90:
    print("Grade: A")
```

```
elif score >= 80:
    print("Grade: B")
```

Output:

Grade: B

## 1.4 The else Statement

The else statement catches anything that isn't caught by the preceding conditions. It provides a fallback block of code to execute when all previous conditions are False.

**Syntax:**

```
if condition1:
    # Code to execute if condition1 is True
elif condition2:
    # Code to execute if condition2 is True
else:
    # Code to execute if all above conditions are False
```

**Example:**

```
temperature = 30

if temperature > 30:
    print("It's a hot day.")
elif temperature > 20:
    print("It's a nice day.")
```

```
else:
    print("It's cold.")
```

Output:

It's a nice day.

## 1.5 Nesting Conditional Statements

Nesting involves placing one conditional statement inside another. This allows for more granular decision-making based on multiple conditions. Nested conditionals are particularly useful when you need to check multiple conditions in a hierarchical manner.

**Understanding Nested Conditionals:**

A nested conditional occurs when an if, elif, or else statement contains another conditional statement within its block. This creates a hierarchy of decisions where the inner condition is only evaluated if the outer condition is True.

Key points about nested conditionals:

- They allow for complex decision trees
- Each level of nesting is indented further
- Can handle interdependent conditions
- Should be used judiciously to maintain code readability

**Syntax:**

```
if condition1:
    if condition2:
        # Code to execute if both condition1 and condition2 are True
    else:
```

```
        # Code to execute if condition1 is True but condition2 is False
else:
    # Code to execute if condition1 is False
```

**Example:**

```
age = 25
has_license = True

if age >= 18:
    if has_license:
        print("You can drive.")
    else:
        print("You need a driver's license.")
else:
    print("You are too young to drive.")
```

Output:

You can drive.

## 1.6 Logical Operators in Conditional Statements

Logical operators allow combining multiple conditions within a single if statement. Python supports three primary logical operators:

and: Returns True if both conditions are True. or: Returns True if at least one condition is True. not: Inverts the boolean value of a condition.

**Syntax:**

```
if condition1 and condition2:
    # Code executes if both conditions are True

if condition1 or condition2:
    # Code executes if at least one condition is True

if not condition:
    # Code executes if condition is False
```

**Examples:**

```
# Using 'and'
age = 25
has_license = True

if age >= 18 and has_license:
    print("You can drive.")
```

Output:

You can drive.

```
# Using 'or'
day = "Sunday"

if day == "Saturday" or day == "Sunday":
    print("It's the weekend!")
```

Output:

It's the weekend!

```
# Using 'not'
is_raining = False

if not is_raining:
    print("You don't need an umbrella today.")
```

Output:

You don't need an umbrella today.

## 1.7 Comparison Operators

Conditional statements often involve comparison operators to evaluate relationships between variables or values. Python supports the following comparison operators:

- ==: Equal to
- !=: Not equal to
- >: Greater than
- <: Less than
- >=: Greater than or equal to
- <=: Less than or equal to

Examples:

```
x = 10
y = 20

```

```
if x < y:
    print("x is less than y.")
```

Output:

x is less than y.

## 1.8 The Ternary Conditional Operator

Python provides a concise way to perform conditional assignments using the ternary operator. This allows you to assign a value based on a condition in a single line.

**Syntax:**

```
variable = value_if_true if condition else value_if_false
```

**Example:**

```
age = 18
status = "Adult" if age >= 18 else "Minor"
print(status)
```

Output:

Adult

## 1.9 Advanced Conditional Constructs

### Using Conditional Statements in Functions

Conditional statements are often used within functions to perform different actions based on input parameters.

Example:

```
def categorize_number(num):
    if num > 0:
        return "Positive"
    elif num < 0:
        return "Negative"
    else:
        return "Zero"

print(categorize_number(10))  # Output: Positive
print(categorize_number(-5))  # Output: Negative
print(categorize_number(0))   # Output: Zero
```

### Conditional Statements with Lists

You can use conditional statements within list operations, such as list comprehensions, to create lists based on conditions.

### Example:

```
numbers = [1, 2, 3, 4, 5]
squares_of_even = [x**2 for x in numbers if x % 2 == 0]
print(squares_of_even)  # Output: [4, 16]
```

## 1.10 Best Practices for Using Conditional Statements

**Keep Conditions Simple:**

Avoid overly complex conditions. If a condition becomes too complicated, consider breaking it down into smaller, named boolean variables for clarity.

```
# Complex condition
if (user.is_active and user.is_verified) or user.is_admin:
    perform_action()

# Simplified with variables
is_active_and_verified = user.is_active and user.is_verified
is_admin = user.is_admin
if is_active_and_verified or is_admin:
    perform_action()
```

**Use Meaningful Variable Names:**

Clear and descriptive variable names make conditional statements easier to understand.

```
# Less clear
if a and b:
    execute()

# More clear
if user.is_active and user.has_subscription:
    execute()
```

### Leverage elif for Multiple Conditions:

Use elif to handle multiple exclusive conditions instead of multiple if statements, which can prevent unintended multiple executions.

```
# Using multiple 'if' statements
if score > 90:
    grade = 'A'
if score > 80:
    grade = 'B'

# Using 'elif'
if score > 90:
    grade = 'A'
elif score > 80:
    grade = 'B'
```

### Avoid Deep Nesting:

Deeply nested conditionals can make code hard to read and maintain. Consider using functions, early returns, or other control structures to flatten the code.

```
# Deeply nested
if condition1:
    if condition2:
        if condition3:
            execute()

# Flattened with early returns
if not condition1:
    return
if not condition2:
    return
```

```
if not condition3:
    return
execute()
```

**Use Boolean Expressions Directly:**

Instead of comparing boolean variables to True or False, use them directly in conditions.

```
# Less Pythonic
if is_valid == True:
    proceed()

# More Pythonic
if is_valid:
    proceed()
```

## 1.11 Common Errors and How to Avoid Them

**Using Assignment (=) Instead of Comparison (==):**

Accidentally using = in conditions leads to syntax errors or unintended behavior.

```
# Incorrect
if x = 10:
    print("x is 10")

# Correct
if x == 10:
    print("x is 10")
```

### Not Accounting for All Possibilities:

Failing to include an else clause can lead to unexpected behavior if none of the conditions are met.

```
# Potential issue
if score > 90:
    grade = 'A'
elif score > 80:
    grade = 'B'
# What if score <= 80?

# Improved
if score > 90:
    grade = 'A'
elif score > 80:
    grade = 'B'
else:
    grade = 'C'
```

### Overusing Nested Conditionals:

Deep nesting can make code difficult to follow. Refactor using helper functions or logical operators.

### Ignoring Operator Precedence:

Misunderstanding how Python evaluates complex logical expressions can lead to bugs. Use parentheses to clarify intended evaluation order.

```
# Potential ambiguity
if condition1 and condition2 or condition3:
    execute()
```

```

# Clarified with parentheses
if (condition1 and condition2) or condition3:
    execute()
```

## 1.12 Practical Examples

### 1.12.1 Example 1: Checking User Eligibility

Let's examine a practical example that demonstrates using nested conditional statements to check user eligibility based on multiple criteria. This example shows how to implement a simple permission system that considers both age and authorization status.

```
def check_eligibility(age, has_permission):
    if age >= 18:
        if has_permission:
            return "Eligible to participate."
        else:
            return "Requires permission."
    else:
        return "Not eligible due to age."

# Usage
print(check_eligibility(20, True))   # Output: Eligible to participate.
print(check_eligibility(16, False))  # Output: Not eligible due to age.
print(check_eligibility(18, False))  # Output: Requires permission.
```

### 1.12.2 Example 2: Temperature Converter

Let's look at another example that demonstrates how to use conditional statements for temperature classification. This example shows how to create a function that categorizes temperatures into different comfort levels using multiple elif statements.

```
def temperature_converter(temp_celsius):
    if temp_celsius < 0:
        return f"{temp_celsius}řC is freezing."
    elif temp_celsius < 25:
        return f"{temp_celsius}řC is cold."
    elif temp_celsius < 35:
        return f"{temp_celsius}řC is warm."
    else:
        return f"{temp_celsius}řC is hot."

# Usage
print(temperature_converter(10))  # Output: 10řC is cold.
print(temperature_converter(30))  # Output: 30řC is warm.
print(temperature_converter(40))  # Output: 40řC is hot.
```

### 1.12.3 Example 3: Simple Login System

This example demonstrates a simple login system implementation in Python. The code showcases the use of nested conditional statements (`if-else`) to perform basic user authentication.

The `login` function takes two parameters:

- `username`: The user's input username
- `password`: The user's input password

The function compares these inputs against hardcoded credentials (a stored username "admin" and password "password123"). Through nested conditional statements, it performs the following checks:

1. First, it verifies if the input username matches the stored username
2. If the username matches, it then checks if the password is correct
3. Based on these checks, it returns appropriate messages:
    - "Login successful" for correct credentials
    - "Incorrect password" for wrong password but correct username
    - "Username not found" for incorrect username

The example demonstrates not only conditional logic but also shows how authentication systems typically work in a simplified form. Note that in real-world applications, storing passwords in plain text is not secure - passwords should always be hashed and properly encrypted.

```
def login(username, password):
    stored_username = "admin"
    stored_password = "password123"

    if username == stored_username:
        if password == stored_password:
            return "Login successful."
        else:
            return "Incorrect password."
    else:
        return "Username not found."

# Usage
print(login("admin", "password123"))  # Output: Login successful.
print(login("admin", "wrongpass"))    # Output: Incorrect password.
print(login("user", "password123"))   # Output: Username not found.
```

## 1.13 Advanced Conditional Constructs

### 1.13.1 Using match Statement (Python 3.10+)

Python 3.10 introduced the match statement, a form of structural pattern matching that can be seen as an advanced form of conditional statements.

textbfSyntax:

```
match variable:
    case pattern1:
        # Code block for pattern1
    case pattern2:
        # Code block for pattern2
    case _:
        # Default case
```

textbfExample:

```
def http_status_message(status_code):
    match status_code:
        case 200:
            return "OK"
        case 404:
            return "Not Found"
        case 500:
            return "Internal Server Error"
        case _:
            return "Unknown Status"

# Usage
print(http_status_message(200))  # Output: OK
print(http_status_message(403))  # Output: Unknown Status
```

## 1.14 Chapter Summary

Conditional statements are a fundamental aspect of Python programming, allowing for dynamic decision-making and control over the flow of execution. This chapter covered the essential components of conditional statements, including if, elif, and else, as well as logical and comparison operators. These tools enable developers to create responsive and adaptable code that can handle a variety of scenarios.

We explored the syntax and usage of basic and nested conditional statements, logical operators, and the ternary operator for concise conditional assignments. Additionally, we introduced the match statement, a powerful feature in Python 3.10 for structural pattern matching.

By following best practicessuch as simplifying conditions, avoiding deep nesting, and using clear variable namesyou can write more readable and maintainable code. Understanding and effectively applying these concepts will enhance your ability to develop robust and efficient Python programs.

This chapter also provided practical examples and exercises to reinforce your understanding and help you apply conditional statements in real-world programming tasks. Mastery of these constructs is crucial for any Python developer aiming to write effective and efficient code.

## 1.15 Exercises on Chapter 4

### Question 1. Grade Assignment

Write a program that assigns a grade based on a score:
90 and above: A
80 to 89: B
70 to 79: C
60 to 69: D
Below 60: F
Input: 85
Output: B

## Question 2. Leap Year Checker

Write a program to check if a given year is a leap year.
Input: 2024
Output: Leap year

## Question 3. Triangle Validity

Given three angles of a triangle, check if the triangle is valid (sum of angles = 180).
Input: 60, 60, 60
Output: Valid triangle

## Question 4. Number Comparison

Write a program that takes three numbers as input and prints the smallest of them.
Input: 5, 3, 8
Output: 3

## Question 5. Check Divisibility

Write a program to check if a number is divisible by both 3 and 5.
Input: 15
Output: Divisible by both

## Question 6. Password Checker

Write a program that checks if a password matches a predefined value. If it matches, print "Access Granted", otherwise print "Access Denied".
Input: password123
Output: Access Granted

## Question 7. Check Leap Year

Write a program that checks if a given year is a leap year.
**Input:** A year (integer).

**Conditions:**

- A year is a leap year if:
  - It is divisible by 4 but not divisible by 100, **or**
  - It is divisible by 400.

**Output:** Print whether the given year is a leap year (e.g., "2024 is a leap year") or not (e.g., "2023 is not a leap year").

## Question 8. Grade Classification

Write a program that classifies a student's score into grades.
**Input:** A student's score (integer between 0 and 100).
**Conditions:** Classify the score as:

- 90100: `A`
- 8089: `B`
- 7079: `C`
- 6069: `D`
- Below 60: `F`

**Output:** Print the grade (e.g., "Score: 85, Grade: B").

## Question 9. Largest of Three Numbers

Write a program that finds the largest number among three given numbers.
**Input:** Three numbers (integers or floats).
**Conditions:** Compare the three numbers using conditionals to find the largest.
**Output:** Print the largest number (e.g., "The largest number is 42.7").

## Question 10. Even-Odd and Divisibility

Write a program that determines if a number is even or odd and checks its divisibility by 3 and 5.
**Input:** A single number (integer).

**Conditions:**

- Check if the number is even or odd.
- Check if the number is divisible by 3 or 5.

**Output:** Print messages such as "The number is even and divisible by 3."

## Question 11. Triangle Validity

Write a program that checks if three given sides form a valid triangle.
**Input:** Three side lengths (positive numbers).
**Conditions:**

- A triangle is valid if the sum of any two sides is greater than the third side.

**Output:** Print whether the triangle is valid (e.g., "Valid triangle") or not (e.g., "Invalid triangle").

## Question 12. Number Guessing Game

Write a number guessing game where the user guesses a secret number between 1 and 100.
**Input:** User guesses a number (integer).
**Conditions:**

- Compare the guess with a secret number (hardcoded or randomly generated).
- Provide hints like "Too high" or "Too low."
- Continue until the user guesses correctly.

**Output:** Print hints and "Correct!" when the user guesses the number.

## Question 13. FizzBuzz with Custom Ranges

Write a program that prints "Fizz," "Buzz," or "FizzBuzz" for numbers in a user-defined range.
**Input:** Start and end values of a range (integers).
**Conditions:** For each number in the range:

- Print "Fizz" if divisible by 3.
- Print "Buzz" if divisible by 5.

- Print "FizzBuzz" if divisible by both 3 and 5.
- Otherwise, print the number.

**Output:** A list of outputs like: `1, 2, Fizz, 4, Buzz, Fizz...`.

## Question 14. Voting Eligibility

Write a program that checks if a person is eligible to vote based on age and citizenship.
**Input:** Age (integer) and citizenship status (boolean or string like "yes"/"no").
**Conditions:**

- Eligible if age $\geq 18$ **and** citizenship is true/yes.

**Output:** Print whether the person is eligible (e.g., "Eligible to vote") or not (e.g., "Not eligible to vote").

## Question 15. Password Strength Checker

Write a program that checks the strength of a given password.
**Input:** A password (string).
**Conditions:**

- A strong password must have:
  - At least 8 characters.
  - At least one number.
  - At least one special character (`@`, `#`, `!`, etc.).

**Output:** Print "Strong password" if all conditions are met, otherwise list weaknesses (e.g., "Password must include at least one special character").

## Question 16. Categorize Age Group

Write a program that categorizes a person into an age group based on their age.
**Input:** A person's age (integer).
**Conditions:**

- 012: "Child"
- 1319: "Teen"
- 2059: "Adult"
- 60+: "Senior"

**Output:** Print the age group (e.g., "Age: 25, Category: Adult").

## Question 17. Calculate Electricity Bill

Write a program that calculates an electricity bill based on the number of units consumed.
**Input:** Units of electricity consumed (integer).
**Conditions:**

- 0100 units: 5/unit.
- 101200 units: 7/unit.
- Above 200 units: 10/unit.

**Output:** Print the total bill (e.g., "Total bill: 950").

## Question 18. Palindrome Check

Write a program that checks if a given string is a palindrome.
**Input:** A string.
**Conditions:**

- A string is a palindrome if it reads the same forwards and backwards.

**Output:** Print whether the string is a palindrome (e.g., "madam is a palindrome") or not (e.g., "hello is not a palindrome").

## Question 19. Number Categorization

Write a program that categorizes a number as positive, negative, zero, even, or odd.
**Input:** A number (integer).
**Conditions:**

- Check if the number is:
  - Positive, negative, or zero.
  - Even or odd.

**Output:** Print messages like "Positive and Odd" or "Negative and Even."

## Question 20. Traffic Light Simulator

Write a program that simulates traffic light actions based on input color.
**Input:** Traffic light color ("red," "yellow," "green").
**Conditions:**

- "red": Print "Stop."
- "yellow": Print "Slow down."
- "green": Print "Go."

**Output:** Print the appropriate action (e.g., "Light: Red, Action: Stop").

## Question 21. BMI Calculator

Write a program that calculates BMI and classifies it into categories like underweight, normal, or obese.
**Input:** Weight (kg) and height (m).
**Conditions:**

- BMI is calculated as `BMI = weight / (height ** 2)`.
- Classify BMI as:
  - Below 18.5: "Underweight."
  - 18.524.9: "Normal weight."
  - 2529.9: "Overweight."
  - 30+: "Obese."

**Output:** Print the BMI value and category (e.g., "BMI: 22.4, Normal weight").