Control Flow in Python

Learning Objectives

By the end of this section, you will be able to:

- Write conditional statements using `if`, `elif`, and `else`
- Create and use `for` and `while` loops
- Control program flow with `break`, `continue`, and `pass` statements
- Use the new `match` statement (Python 3.10+) for pattern matching

Conditional Statements

Conditional statements allow your program to make decisions based on certain conditions.

Basic `if` Statement

The `if` statement evaluates a condition and executes a block of code if the condition is `True`.

```python
# Basic if statement syntax
if condition:
    # code to execute if condition is True
    # This indented block belongs to the if statement
```

Example:

```python
age = 20

if age >= 18:
    print("You are an adult.")
    print("You can vote.")

print("This will print regardless of the condition.")
```

`if-else` Statement

The `else` clause provides an alternative block of code to execute when the condition is `False`.

```python
# if-else syntax
if condition:
    # code to execute if condition is True
else:
    # code to execute if condition is False
```

Example:

```
age = 15

if age >= 18:
    print("You are an adult.")
    print("You can vote.")
else:
    print("You are a minor.")
    print("You cannot vote yet.")
```

`if-elif-else` Statement

The `elif` (short for "else if") statement allows you to check multiple conditions in sequence.

```
# if-elif-else syntax
if condition1:
    # code to execute if condition1 is True
elif condition2:
    # code to execute if condition1 is False and condition2 is True
elif condition3:
    # code to execute if condition1 and condition2 are False and
condition3 is True
else:
    # code to execute if all conditions are False
```

Example:

```
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}")
```

Nested Conditionals

You can place conditional statements inside other conditional statements.

```
weather = "sunny"
temperature = 28  # Celsius

if weather == "sunny":
    print("It's a sunny day!")
    if temperature > 25:
        print("It's hot outside.")
    else:
        print("It's warm but pleasant.")
elif weather == "rainy":
    print("Don't forget your umbrella!")
else:
    print("Check the weather forecast.")
```

Conditional Expressions (Ternary Operator)

Python supports a compact way to write simple if-else statements:

```
# Syntax: value_if_true if condition else value_if_false
age = 20
status = "adult" if age >= 18 else "minor"
print(status)  # Output: adult
```

Loops

Loops allow you to execute a block of code multiple times.

`for` Loops

A `for` loop is used to iterate over a sequence (like a list, tuple, string, or range).

```
# Basic for loop syntax
for item in sequence:
    # code to execute for each item
```

Iterating Through Ranges

The `range()` function generates a sequence of numbers:

```
# range(stop): 0 to stop-1
for i in range(5):
    print(i)  # Outputs: 0, 1, 2, 3, 4

# range(start, stop): start to stop-1
for i in range(2, 6):
    print(i)  # Outputs: 2, 3, 4, 5
```

```python
# range(start, stop, step): with custom step
for i in range(0, 10, 2):
    print(i)  # Outputs: 0, 2, 4, 6, 8
```

Iterating Through Collections

```python
# Iterating through a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(f"I like {fruit}s")

# Iterating through a string
for char in "Python":
    print(char)

# Iterating through a dictionary
student = {"name": "Alice", "age": 20, "grade": "A"}

# Iterating through keys (default)
for key in student:
    print(f"{key}: {student[key]}")

# Iterating through key-value pairs
for key, value in student.items():
    print(f"{key}: {value}")
```

Using `enumerate()` for Index and Value

```python
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(f"{index + 1}: {fruit}")
```

`while` Loops

A `while` loop executes a block of code as long as a condition is `True`.

```python
# Basic while loop syntax
while condition:
    # code to execute while condition is True
    # Make sure the condition will eventually become False
```

Example:

```
count = 1
while count <= 5:
    print(count)
    count += 1  # Don't forget to update the condition variable!
```

Infinite Loops

Be careful not to create infinite loops, which continue indefinitely because the condition never becomes `False`:

```
# WARNING: This is an infinite loop
# while True:
#     print("This will run forever!")

# Proper infinite loop with a way to exit
while True:
    response = input("Continue? (yes/no): ")
    if response.lower() == "no":
        break  # Exit the loop
```

Loop Control Statements

The `break` Statement

The `break` statement immediately terminates the loop it's in:

```
# Using break in a for loop
for i in range(10):
    if i == 5:
        print("Found 5! Breaking the loop.")
        break
    print(i)
# Outputs: 0, 1, 2, 3, 4, "Found 5! Breaking the loop."

# Using break in a while loop
count = 0
while True:
    count += 1
    if count > 5:
        break
    print(count)
# Outputs: 1, 2, 3, 4, 5
```

The `continue` Statement

The `continue` statement skips the current iteration and moves to the next one:

```
# Using continue in a for loop
for i in range(10):
    if i % 2 == 0:  # Skip even numbers
        continue
    print(i)
# Outputs: 1, 3, 5, 7, 9

# Using continue in a while loop
count = 0
while count < 10:
    count += 1
    if count % 2 == 0:  # Skip even numbers
        continue
    print(count)
# Outputs: 1, 3, 5, 7, 9
```

The `pass` Statement

The `pass` statement is a no-operation placeholder. It does nothing but avoids syntax errors when a statement is required:

```
# Using pass as a placeholder
for i in range(5):
    if i == 3:
        pass  # TODO: Add something here later
    else:
        print(i)
```

The `match` Statement (Python 3.10+)

The `match` statement (introduced in Python 3.10) provides pattern matching similar to switch statements in other languages, but more powerful.

Basic Match Statement

```
# Basic match statement syntax
match value:
    case pattern1:
        # code if value matches pattern1
    case pattern2:
        # code if value matches pattern2
    case _:
        # default case if no pattern matches
```

Example:

```python
def day_type(day):
    match day.lower():
        case "monday":
            return "Start of work week"
        case "tuesday" | "wednesday" | "thursday":
            return "Midweek"
        case "friday":
            return "End of work week"
        case "saturday" | "sunday":
            return "Weekend"
        case _:
            return "Invalid day"

print(day_type("Monday"))  # Start of work week
print(day_type("Saturday"))  # Weekend
print(day_type("Holiday"))  # Invalid day
```

Pattern Matching with Structures

The `match` statement can also match against structured patterns:

```python
def process_command(command):
    match command.split():
        case ["quit"]:
            return "Exiting program"
        case ["load", filename]:
            return f"Loading file: {filename}"
        case ["save", filename]:
            return f"Saving file: {filename}"
        case ["search", *keywords]:
            return f"Searching for: {' '.join(keywords)}"
        case _:
            return "Unknown command"

print(process_command("quit"))  # Exiting program
print(process_command("load data.txt"))  # Loading file: data.txt
print(process_command("search python tutorial"))  # Searching for: python
tutorial
```

Pattern Matching with Guards

You can add conditions to patterns using guards:

```python
def check_temperature(temp):
    match temp:
        case int() | float() as t if t < 0:
            return "Freezing"
        case int() | float() as t if 0 <= t < 20:
                return "Cold"
            case int() | float() as t if 20 <= t < 30:
                return "Pleasant"
            case int() | float() as t if t >= 30:
                return "Hot"
            case _:
                return "Invalid temperature"

print(check_temperature(-5))    # Freezing
print(check_temperature(15))    # Cold
print(check_temperature(25))    # Pleasant
print(check_temperature(35))    # Hot
print(check_temperature("35"))  # Invalid temperature
```

Practice Exercises

Exercise 1: Conditional Statements

Create a program that:

1. Asks the user for their age
2. Determines and prints which category they fall into:
   - Child (0-12)
   - Teenager (13-19)
   - Adult (20-64)
   - Senior (65+)
3. If the age is negative or over 120, print an error message

```python
# Exercise 1 template
age = int(input("Enter your age: "))

# Your code here
```

Exercise 2: Nested Loops

Create a program that:

1. Prints a multiplication table for numbers 1 through 5
2. Format the output so it's neatly aligned in rows and columns

```python
# Exercise 2 template
# Your code here
```

Exercise 3: Loop Control Statements

Create a program that:

1. Asks the user for numbers, one at a time
2. Continues asking until the user enters 0 (use a while loop)
3. Skips negative numbers (using continue)
4. Exits immediately if the user enters 999 (using break)
5. At the end, prints the sum of all positive numbers entered

```python
# Exercise 3 template
total = 0

# Your code here
```

Exercise 4: Match Statement (Python 3.10+)

Create a simple calculator program that:

1. Asks the user for two numbers
2. Asks for an operation (add, subtract, multiply, divide)
3. Uses a match statement to perform the requested operation
4. Handles division by zero appropriately
5. Returns an error for unknown operations

```python
# Exercise 4 template (requires Python 3.10+)
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
operation = input("Enter operation (add/subtract/multiply/divide): ")

# Your code here
```

Key Takeaways

- Conditional statements allow your program to make decisions based on conditions
- Loops provide a way to repeat code execution
- `break` exits a loop completely, `continue` skips to the next iteration, and `pass` is a placeholder
- The `match` statement (Python 3.10+) offers powerful pattern matching capabilities
- Always ensure that loops have a way to terminate to avoid infinite loops
- Indentation is critical in Python for defining the structure of conditional statements and loops

Next Steps

Now that you understand Python's control flow mechanisms, you're ready to explore data structures like lists, tuples, and dictionaries!