Python Programming Guide - UNIT III: Control Flow and Functions

1. Conditional Statements

Conditional statements allow you to execute different code blocks based on conditions. They are essential for decision-making in programs.

if Statement

The `if` statement executes a block of code only if a condition is true.

```
# Basic if statement
age = 18
if age >= 18:
    print("You are an adult")

# Single line if (not recommended for readability)
if age >= 18: print("Adult")

# Boolean condition
is_student = True
if is_student:
    print("Student discount applied")

# Practical example
temperature = 30
if temperature > 25:
    print("It's hot outside")
    print("Drink water")

# if with compound conditions
score = 85
attendance = 95
if score >= 80 and attendance >= 90:
    print("Grade A: Excellent performance!")
```

if-else Statement

The `if-else` statement executes one block if the condition is true, and a different block if it's false.

```
# Basic if-else
```

```
age = 15
if age >= 18:
    print("You can vote")
else:
    print("You cannot vote yet")

# Checking login credentials
username = "alice"
password = "secure123"

entered_username = "alice"
entered_password = "secure123"

if entered_username == username and entered_password == password:
    print("Login successful")
else:
    print("Invalid credentials")

# Checking even or odd
number = 7
if number % 2 == 0:
    print(f"{number} is even")
else:
    print(f"{number} is odd")

# Processing user input
score = int(input("Enter your score: "))
if score >= 50:
    print("You passed")
else:
    print("You failed")
```

if-elif-else Statement

The `if-elif-else` statement allows checking multiple conditions.

```
# Grade assignment
score = 78
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"Score: {score}, Grade: {grade}")

# Multiple elif conditions
age = 25
if age < 13:
    category = "Child"
elif age < 18:
    category = "Teenager"
elif age < 60:
    category = "Adult"
else:
```

```
    category = "Senior"

print(f"Age: {age}, Category: {category}")

# Complex conditional logic
temperature = 35
if temperature < 0:
    weather = "Freezing"
elif temperature < 10:
    weather = "Cold"
elif temperature < 20:
    weather = "Cool"
elif temperature < 30:
    weather = "Warm"
else:
    weather = "Hot"

print(f"Temperature: {temperature}°C - {weather}")
```

Real-World Application: Student Result Analysis

```
class StudentResultAnalyzer:
    """Analyze and report student results."""

    # Grade boundaries
    GRADE_A_MIN = 90
    GRADE_B_MIN = 80
    GRADE_C_MIN = 70
    GRADE_D_MIN = 60
    GRADE_F_MIN = 0

    # Pass mark
    PASS_MARK = 60

    @staticmethod
    def get_grade(marks):
        """Determine grade based on marks."""
        if marks >= StudentResultAnalyzer.GRADE_A_MIN:
            return 'A'
        elif marks >= StudentResultAnalyzer.GRADE_B_MIN:
            return 'B'
        elif marks >= StudentResultAnalyzer.GRADE_C_MIN:
            return 'C'
        elif marks >= StudentResultAnalyzer.GRADE_D_MIN:
            return 'D'
        else:
            return 'F'

    @staticmethod
    def get_remarks(marks):
        """Get remarks based on performance."""
        if marks >= 95:
            return "Outstanding!"
        elif marks >= 90:
            return "Excellent!"
        elif marks >= 80:
            return "Very Good"
        elif marks >= 70:
            return "Good"
        elif marks >= 60:
            return "Satisfactory"
```

```
                else:
                    return "Needs Improvement"

            @staticmethod
            def analyze_result(student_name, marks):
                """Provide complete result analysis."""
                grade = StudentResultAnalyzer.get_grade(marks)
                remarks = StudentResultAnalyzer.get_remarks(marks)
                status = "Pass" if marks >= StudentResultAnalyzer.PASS_MARK
else "Fail"

                print(f"\n{'='*50}")
                print(f"RESULT ANALYSIS - {student_name.upper()}")
                print(f"{'='*50}")
                print(f"Marks: {marks}")
                print(f"Grade: {grade}")
                print(f"Status: {status}")
                print(f"Remarks: {remarks}")
                print(f"{'='*50}")

        # Usage
        students = [
            ("Alice", 95),
            ("Bob", 78),
            ("Charlie", 55),
            ("Diana", 88)
        ]

        for name, marks in students:
            StudentResultAnalyzer.analyze_result(name, marks)
```

Output:

```
==================================================
RESULT ANALYSIS - ALICE
==================================================
Marks: 95
Grade: A
Status: Pass
Remarks: Outstanding!
==================================================

... (similar for other students)
```

---

2. Looping Statements

Loops allow you to execute a block of code multiple times, which is essential for repetitive tasks.

for Loop

The `for` loop iterates over sequences like lists, strings, or ranges.

```
        # Loop through a list
        fruits = ['apple', 'banana', 'cherry']
        for fruit in fruits:
            print(fruit)
```

```python
# Loop with index using enumerate()
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")
# Output: 0: apple / 1: banana / 2: cherry

# Loop through a range
for i in range(5):
    print(i)  # Output: 0 1 2 3 4

# Range with start and stop
for i in range(1, 6):
    print(i)  # Output: 1 2 3 4 5

# Range with step
for i in range(0, 10, 2):
    print(i)  # Output: 0 2 4 6 8

# Reverse range
for i in range(5, 0, -1):
    print(i)  # Output: 5 4 3 2 1

# Loop through string
word = "Python"
for letter in word:
    print(letter)  # P y t h o n

# Nested loops
for i in range(1, 4):
    for j in range(1, 4):
        print(f"({i},{j})", end=" ")
    print()
# Output: (1,1) (1,2) (1,3)
#         (2,1) (2,2) (2,3)
#         (3,1) (3,2) (3,3)

# Loop through dictionary
person = {'name': 'Alice', 'age': 25, 'city': 'NYC'}
for key in person:
    print(f"{key}: {person[key]}")

for key, value in person.items():
    print(f"{key}: {value}")
```

while Loop

The `while` loop continues executing while a condition is true.

```python
# Basic while loop
count = 0
while count < 5:
    print(count)
    count += 1

# Loop with user input (until valid input)
while True:
    user_input = input("Enter 'quit' to exit: ")
    if user_input.lower() == 'quit':
        break
    print(f"You entered: {user_input}")
```

```
        # Countdown timer
        count = 10
        while count > 0:
            print(f"Time remaining: {count} seconds")
            count -= 1
        print("Blast off!")

        # Validation loop
        password = ""
        while len(password) < 8:
            password = input("Enter password (min 8 characters): ")
            if len(password) < 8:
                print("Password too short. Try again.")

        print("Password accepted!")

        # While loop with condition
        number = 1
        while number <= 10 and number % 2 == 0:
            print(number)
            number += 2
```

Real-World Application: User Authentication System

```
        class AuthenticationSystem:
            """Manage user authentication."""

            MAX_ATTEMPTS = 3
            CORRECT_PASSWORD = "secure123"

            @staticmethod
            def login():
                """Handle user login with retry attempts."""
                print("\n" + "="*50)
                print("USER LOGIN SYSTEM")
                print("="*50)

                attempts = 0
                while attempts < AuthenticationSystem.MAX_ATTEMPTS:
                    password = input("Enter password: ")

                    if password == AuthenticationSystem.CORRECT_PASSWORD:
                        print("✓ Login successful!")
                        return True
                    else:
                        attempts += 1
                        remaining = AuthenticationSystem.MAX_ATTEMPTS -
attempts
                        if remaining > 0:
                            print(f"✗ Incorrect password. {remaining}
attempts remaining.")
                        else:
                            print("✗ Maximum attempts exceeded. Account
locked.")
                            return False

                return False

        # Usage
        # AuthenticationSystem.login()
```

---

3. Loop Control Statements

Loop control statements modify the flow of loops: `break`, `continue`, and `pass`.

break Statement

The `break` statement exits the loop immediately.

```
# Break on condition
for i in range(10):
    if i == 5:
        break
    print(i)
# Output: 0 1 2 3 4 (stops before 5)

# Break with while loop
count = 0
while True:
    if count == 3:
        break
    print(count)
    count += 1
# Output: 0 1 2

# Break in nested loops
for i in range(3):
    for j in range(3):
        if j == 1:
            break  # Breaks inner loop only
        print(f"({i},{j})", end=" ")
    print()  # Newline after each iteration of outer loop

# Search operation with break
numbers = [1, 3, 5, 7, 9, 2, 4]
target = 7
for num in numbers:
    if num == target:
        print(f"Found {target}!")
        break
else:
    print(f"{target} not found")
```

continue Statement

The `continue` statement skips the rest of the current iteration and moves to the next one.

```
# Skip even numbers
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
# Output: 1 3 5 7 9 (all odd numbers)

# Skip invalid inputs
items = [1, -2, 3, -4, 5]
```

```
for item in items:
    if item < 0:
        continue
    print(f"Processing: {item}")
# Output: Processing: 1 / 3 / 5

# Continue in nested loops
for i in range(1, 4):
    for j in range(1, 4):
        if j == 2:
            continue
        print(f"({i},{j})", end=" ")
    print()

# Filter data using continue
numbers = [10, 15, 20, 25, 30, 35, 40]
for num in numbers:
    if num % 5 == 0 and num < 30:
        continue
    print(num)
```

pass Statement

The `pass` statement is a null operation — it does nothing. It's useful as a placeholder.

```
# Placeholder for unimplemented functions
def future_function():
    pass

# Placeholder in conditional
if True:
    pass  # To be implemented later

# Placeholder in loop
for i in range(5):
    if i == 3:
        pass
    else:
        print(i)

# Placeholder in class definition
class MyClass:
    pass

# Placeholder in try-except
try:
    result = 10 / 0
except ZeroDivisionError:
    pass  # Ignore division by zero errors

# Real use case: ignore specific items
for item in range(10):
    if item == 5:
        pass  # Skip 5 (will be handled later)
    else:
        print(item)
```

Real-World Application: Data Filtering and Processing

```python
class DataProcessor:
    """Process and filter data."""

    @staticmethod
    def filter_valid_ages(ages):
        """Filter valid ages (18-120)."""
        MIN_AGE = 18
        MAX_AGE = 120

        valid_ages = []
        for age in ages:
            if age < MIN_AGE or age > MAX_AGE:
                continue  # Skip invalid ages
            valid_ages.append(age)
        return valid_ages

    @staticmethod
    def find_student(student_list, target_id):
        """Find student by ID."""
        for student in student_list:
            if student['id'] == target_id:
                return student
            # If not found, continue to next student
        return None  # No student found

    @staticmethod
    def process_batch(data, should_stop_callback):
        """Process data with early stopping."""
        results = []
        for item in data:
            if should_stop_callback(item):
                break  # Stop processing
            results.append(item * 2)
        return results

    @staticmethod
    def generate_clean_data(raw_data):
        """Clean data by removing invalid entries."""
        clean_data = []
        for value in raw_data:
            if value is None or value == "":
                continue  # Skip empty values
            if isinstance(value, str) and value.startswith("#"):
                continue  # Skip comments
            clean_data.append(value)
        return clean_data

# Usage
print("Filtering ages...")
ages = [15, 25, 35, 150, 45, -5, 55]
valid = DataProcessor.filter_valid_ages(ages)
print(f"Valid ages: {valid}")

print("\nProcessing data...")
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
result = DataProcessor.process_batch(data, lambda x: x > 5)
print(f"Processed: {result}")
```

4. Functions

Functions are reusable blocks of code that perform specific tasks. They promote code organization, reusability, and maintainability.

Built-in Functions

Python provides many built-in functions that you can use directly.

```
        # String functions
        text = "Hello World"
        print(len(text))            # Output: 11 (length)
        print(text.upper())         # Output: HELLO WORLD
        print(text.lower())         # Output: hello world
        print(text.replace("World", "Python"))  # Output: Hello Python

        # Numeric functions
        print(abs(-5))              # Output: 5 (absolute value)
        print(round(3.7))           # Output: 4 (round)
        print(max([1, 5, 3]))       # Output: 5 (maximum)
        print(min([1, 5, 3]))       # Output: 1 (minimum)
        print(sum([1, 2, 3]))       # Output: 6 (sum)
        print(pow(2, 3))            # Output: 8 (power)

        # Type conversion functions
        print(int("123"))           # Output: 123
        print(float("3.14"))        # Output: 3.14
        print(str(100))             # Output: '100'
        print(bool(1))              # Output: True
        print(list("abc"))          # Output: ['a', 'b', 'c']

        # Iteration functions
        for index, value in enumerate(['a', 'b', 'c']):
            print(f"{index}: {value}")

        zipped = list(zip([1, 2, 3], ['a', 'b', 'c']))
        print(zipped)               # Output: [(1, 'a'), (2, 'b'), (3, 'c')]

        # Functional functions
        numbers = [1, 2, 3, 4, 5]
        squared = list(map(lambda x: x**2, numbers))
        print(squared)              # Output: [1, 4, 9, 16, 25]

        evens = list(filter(lambda x: x % 2 == 0, numbers))
        print(evens)                # Output: [2, 4]

        # Sorting function
        print(sorted([3, 1, 4, 1, 5]))  # Output: [1, 1, 3, 4, 5]
        print(sorted([3, 1, 4, 1, 5], reverse=True))  # Output: [5, 4, 3, 1,
1]
```

User-Defined Functions

You create functions using the `def` keyword.

```
        # Basic function
        def greet():
            print("Hello!")
```

```
greet()  # Call the function

# Function with parameters
def greet_person(name):
    print(f"Hello, {name}!")

greet_person("Alice")       # Output: Hello, Alice!

# Function with return value
def add(a, b):
    return a + b

result = add(5, 3)
print(result)               # Output: 8

# Function with multiple parameters
def calculate_area(length, width):
    """Calculate area of a rectangle."""
    return length * width

area = calculate_area(5, 10)
print(f"Area: {area}")      # Output: Area: 50

# Function with default parameters
def introduce(name, age=25, city="New York"):
    return f"{name} is {age} years old and lives in {city}"

print(introduce("Alice"))
print(introduce("Bob", 30))
print(introduce("Charlie", 35, "Boston"))

# Function with variable number of arguments (*args)
def sum_all(*numbers):
    total = 0
    for num in numbers:
        total += num
    return total

print(sum_all(1, 2, 3))          # Output: 6
print(sum_all(1, 2, 3, 4, 5))    # Output: 15

# Function with keyword arguments (**kwargs)
def print_info(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25, city="NYC")

# Combining positional, *args, and **kwargs
def full_function(name, *scores, **info):
    print(f"Name: {name}")
    print(f"Scores: {scores}")
    print(f"Info: {info}")

full_function("Alice", 85, 90, 95, school="MIT", year=2026)
```

Real-World Application: Grade Calculator

```
class GradeCalculator:
    """Calculate student grades and statistics."""
```

```
            @staticmethod
            def calculate_average(*scores):
                """Calculate average of any number of scores."""
                if not scores:
                    return 0
                return sum(scores) / len(scores)

            @staticmethod
            def get_grade_letter(average):
                """Get letter grade from average."""
                if average >= 90:
                    return 'A'
                elif average >= 80:
                    return 'B'
                elif average >= 70:
                    return 'C'
                elif average >= 60:
                    return 'D'
                else:
                    return 'F'

            @staticmethod
            def generate_report(name, **subject_scores):
                """Generate a detailed grade report."""
                print(f"\nGRADE REPORT - {name}")
                print("=" * 50)

                averages = {}
                overall_scores = []

                for subject, scores in subject_scores.items():
                    avg = GradeCalculator.calculate_average(*scores)
                    grade = GradeCalculator.get_grade_letter(avg)
                    averages[subject] = avg
                    overall_scores.extend(scores)
                    print(f"{subject}: {avg:.2f} ({grade})")

                overall_avg =
GradeCalculator.calculate_average(*overall_scores)
                overall_grade =
GradeCalculator.get_grade_letter(overall_avg)

                print("-" * 50)
                print(f"Overall Average: {overall_avg:.2f}
({overall_grade})")
                print("=" * 50)

                return overall_avg

        # Usage
        GradeCalculator.generate_report(
            "Alice",
            math=[85, 90, 88],
            english=[80, 85, 82],
            science=[92, 88, 90]
        )
```

---

5. Types of Arguments

Functions can accept different types of arguments:

```python
# 1. Positional arguments (order matters)
def introduce(first_name, last_name):
    return f"{first_name} {last_name}"

print(introduce("John", "Doe"))      # Output: John Doe
# print(introduce("Doe", "John"))    # Wrong order, wrong result

# 2. Keyword arguments (order doesn't matter)
def create_user(name, email, age):
    return f"User: {name}, Email: {email}, Age: {age}"

print(create_user(name="Alice", email="alice@example.com", age=25))
print(create_user(age=25, name="Alice", email="alice@example.com"))

# 3. Default arguments
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

print(greet("Alice"))                    # Output: Hello, Alice!
print(greet("Alice", "Hi"))              # Output: Hi, Alice!

# 4. *args (Variable number of positional arguments)
def add_numbers(*args):
    """Add any number of arguments."""
    return sum(args)

print(add_numbers(1, 2, 3))              # Output: 6
print(add_numbers(1, 2, 3, 4, 5))        # Output: 15

# 5. **kwargs (Variable number of keyword arguments)
def build_profile(**kwargs):
    """Build a profile from keyword arguments."""
    profile = {}
    for key, value in kwargs.items():
        profile[key] = value
    return profile

user_profile = build_profile(
    name="Alice",
    age=25,
    city="NYC",
    job="Engineer"
)
print(user_profile)

# 6. Combined arguments
def full_function(name, age, *hobbies, **info):
    """Demonstrate all argument types."""
    print(f"Name: {name}")
    print(f"Age: {age}")
    print(f"Hobbies: {hobbies}")
    print(f"Additional Info: {info}")

full_function(
    "Alice", 25,
    "reading", "gaming", "swimming",
    school="MIT",
    country="USA"
)
```

6. Recursive Functions

Recursive functions call themselves to solve problems by breaking them into smaller subproblems.

```python
# 1. Simple recursion - Factorial
def factorial(n):
    """Calculate factorial recursively: n! = n * (n-1)!"""
    if n == 0 or n == 1:  # Base case
        return 1
    else:
        return n * factorial(n - 1)  # Recursive case

print(f"5! = {factorial(5)}")  # Output: 5! = 120

# 2. Fibonacci sequence
def fibonacci(n):
    """Generate nth Fibonacci number."""
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

print(f"Fibonacci(7) = {fibonacci(7)}")  # Output: Fibonacci(7) = 13

# 3. String reversal
def reverse_string(s):
    """Reverse a string recursively."""
    if len(s) == 0:
        return s
    else:
        return reverse_string(s[1:]) + s[0]

print(reverse_string("Hello"))  # Output: olleH

# 4. Binary search (recursive)
def binary_search(arr, target, left=0, right=None):
    """Search for target in sorted array."""
    if right is None:
        right = len(arr) - 1

    if left > right:
        return -1  # Not found

    mid = (left + right) // 2

    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search(arr, target, mid + 1, right)
    else:
        return binary_search(arr, target, left, mid - 1)

numbers = [1, 3, 5, 7, 9, 11, 13, 15]
print(binary_search(numbers, 7))  # Output: 3

# 5. Power calculation
```

```python
def power(base, exponent):
    """Calculate base^exponent recursively."""
    if exponent == 0:
        return 1
    else:
        return base * power(base, exponent - 1)

print(power(2, 5))  # Output: 32

# 6. Tree traversal (recursive)
def traverse_tree(node):
    """Traverse and print tree nodes."""
    if node is None:
        return
    print(node['value'], end=" ")
    if 'left' in node:
        traverse_tree(node['left'])
    if 'right' in node:
        traverse_tree(node['right'])

# Example tree
tree = {
    'value': 1,
    'left': {
        'value': 2,
        'left': {'value': 4},
        'right': {'value': 5}
    },
    'right': {
        'value': 3,
        'left': {'value': 6},
        'right': {'value': 7}
    }
}

print("Tree traversal: ", end="")
traverse_tree(tree)  # Output: 1 2 4 5 3 6 7
```

Real-World Application: Recursive File Search

```python
import os

class FileSearcher:
    """Search for files recursively in directories."""

    @staticmethod
    def search_files(directory, extension, results=None):
        """Recursively search for files with given extension."""
        if results is None:
            results = []

        try:
            for item in os.listdir(directory):
                path = os.path.join(directory, item)

                if os.path.isfile(path):
                    if path.endswith(extension):
                        results.append(path)

                elif os.path.isdir(path):
                    # Recursive call for subdirectories
```

```
                        FileSearcher.search_files(path, extension,
results)

                except PermissionError:
                    pass  # Skip directories without permission

                return results

            @staticmethod
            def count_files(directory, count=0):
                """Count files recursively."""
                try:
                    for item in os.listdir(directory):
                        path = os.path.join(directory, item)

                        if os.path.isfile(path):
                            count += 1
                        elif os.path.isdir(path):
                            count = FileSearcher.count_files(path, count)

                except PermissionError:
                    pass

                return count

        # Usage (example - would work with actual directories)
        # python_files = FileSearcher.search_files('.', '.py')
        # print(f"Found {len(python_files)} Python files")
```

---

7. Lambda Functions

Lambda functions are small anonymous functions defined with the `lambda` keyword. They're useful for simple, one-time operations.

```
        # Basic lambda function
        add = lambda x, y: x + y
        print(add(5, 3))  # Output: 8

        # Lambda with single argument
        square = lambda x: x ** 2
        print(square(5))  # Output: 25

        # Lambda in function calls (map)
        numbers = [1, 2, 3, 4, 5]
        squared = list(map(lambda x: x**2, numbers))
        print(squared)  # Output: [1, 4, 9, 16, 25]

        # Lambda in filter()
        numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        evens = list(filter(lambda x: x % 2 == 0, numbers))
        print(evens)  # Output: [2, 4, 6, 8, 10]

        # Lambda for sorting
        students = [
            {'name': 'Alice', 'score': 85},
            {'name': 'Bob', 'score': 78},
            {'name': 'Charlie', 'score': 92}
        ]
```

```
        sorted_students = sorted(students, key=lambda s: s['score'],
reverse=True)
        for student in sorted_students:
            print(f"{student['name']}: {student['score']}")

        # Lambda in reduce() (from functools)
        from functools import reduce
        numbers = [1, 2, 3, 4, 5]
        product = reduce(lambda x, y: x * y, numbers)
        print(f"Product: {product}")  # Output: Product: 120

        # Lambda with conditionals
        is_adult = lambda age: "Adult" if age >= 18 else "Minor"
        print(is_adult(20))  # Output: Adult
        print(is_adult(15))  # Output: Minor

        # Practical example: data transformation
        data = [
            {'price': 100, 'quantity': 2},
            {'price': 50, 'quantity': 3},
            {'price': 75, 'quantity': 1}
        ]

        totals = list(map(lambda item: item['price'] * item['quantity'],
data))
        print(totals)  # Output: [200, 150, 75]
```

Real-World Application: Data Processing Pipeline

```
        from functools import reduce

        class DataProcessor:
            """Process data using lambda functions."""

            @staticmethod
            def process_sales_data(sales):
                """Process and analyze sales data."""
                # Filter high-value sales
                high_sales = list(filter(lambda s: s['amount'] > 1000,
sales))

                # Apply discount to high-value sales
                discounted = list(map(lambda s: {**s, 'amount': s['amount']
* 0.9}, high_sales))

                # Calculate total
                total = reduce(lambda acc, s: acc + s['amount'], discounted,
0)

                return {
                    'high_value_sales': len(discounted),
                    'total_revenue': total,
                    'average_sale': total / len(discounted) if discounted
else 0
                }

            @staticmethod
            def filter_and_sort_products(products, min_price=0,
max_price=float('inf')):
                """Filter and sort products by price."""
```

```
                filtered = filter(
                    lambda p: min_price <= p['price'] <= max_price,
                    products
                )
                return sorted(filtered, key=lambda p: p['price'])

        # Usage
        sales_data = [
            {'date': '2026-01-01', 'amount': 1500},
            {'date': '2026-01-02', 'amount': 800},
            {'date': '2026-01-03', 'amount': 2000},
            {'date': '2026-01-04', 'amount': 950},
            {'date': '2026-01-05', 'amount': 1200}
        ]

        result = DataProcessor.process_sales_data(sales_data)
        print(f"\nSales Analysis:")
        print(f"High-value sales (>$1000): {result['high_value_sales']}")
        print(f"Total revenue after discount:
${result['total_revenue']:.2f}")
        print(f"Average sale: ${result['average_sale']:.2f}")
```

Summary of Unit III

This unit covers control flow and functions:

- **Conditional Statements**: if, if-else, if-elif-else for decision-making
- **Looping**: for loops and while loops for repetition
- **Loop Control**: break, continue, pass for controlling loop flow
- **Functions**: Built-in and user-defined functions
- **Arguments**: Positional, keyword, default, *args, and **kwargs
- **Recursion**: Functions calling themselves
- **Lambda Functions**: Anonymous functions for simple operations

---

Practice Exercises for Unit III

1. Create a program that uses all three conditional statement types
2. Write a for loop and while loop that produce the same output
3. Implement functions for common mathematical operations (max, min, average)
4. Create a recursive function to solve a real-world problem
5. Build a data processing pipeline using lambda functions with map and filter

---

Next Unit: Unit IV - Data Structures and File Handling