Chapter 14: Multiple Modules and the RPN Example

A Comprehensive Tutorial

Introduction

This tutorial provides a complete guide to **Chapter 14: Multiple Modules and the RPN Example**, covering modular programming in Python. You will learn how to organize code across multiple files, understand import mechanisms, manage module visibility, and build a comprehensive Reverse Polish Notation (RPN) calculator application across multiple modules.

14.1 Overview of Modules in Python

What is a Module?

A **module** in Python is simply a file containing Python code with a `.py` extension. Modules help organize code by grouping related functions, classes, and variables together.

Benefits of Using Modules

- **Code Reusability**: Write once, use multiple times
- **Namespace Management**: Avoid naming conflicts
- **Code Organization**: Logical separation of functionality
- **Maintainability**: Easier to locate and fix bugs
- **Collaboration**: Team members can work on different modules

Creating a Simple Module

To create a module, simply save Python code in a `.py` file.

Example: math_utils.py

```
# math_utils.py
def add(a, b):
    """Return the sum of a and b"""
    return a + b

def subtract(a, b):
    """Return the difference of a and b"""
    return a - b

def multiply(a, b):
    """Return the product of a and b"""
```

```
    return a * b

def divide(a, b):
    """Return the quotient of a and b"""
    if b == 0:
        return "Error: Division by zero"
    return a / b

PI = 3.14159
```

Using a Module

To use a module, you **import** it into your program:

```
# main.py
import math_utils

result = math_utils.add(10, 5)
print(f"10 + 5 = {result}")

result = math_utils.multiply(3, 4)
print(f"3 * 4 = {result}")

print(f"Value of PI: {math_utils.PI}")
```

Output:

```
10 + 5 = 15
3 * 4 = 12
Value of PI: 3.14159
```

14.2 Simple Two-Module Example

Let's create a practical two-module example demonstrating module interaction.

Module 1: util_module.py

```
# util_module.py
"""Utility functions for string operations"""

def shout(s):
    """Convert string to uppercase and add exclamation"""
    return s.upper() + '!'

def whisper(s):
    """Convert string to lowercase"""
    return s.lower()

def repeat(s, n=2):
    """Repeat string n times"""
    return s * n
```

```
# Module-level variable
greeting = "Hello from util_module"
```

Module 2: main_program.py

```
# main_program.py
"""Main program using util_module"""

import util_module

# Using functions from util_module
message = "Python Programming"
print(util_module.shout(message))
print(util_module.whisper(message))
print(util_module.repeat("ABC", 3))

# Accessing module variable
print(util_module.greeting)
```

Output:

```
PYTHON PROGRAMMING!
python programming
ABCABCABC
Hello from util_module
```

Example: Temperature Converter

Module: temp_converter.py

```
# temp_converter.py
def celsius_to_fahrenheit(celsius):
    """Convert Celsius to Fahrenheit"""
    return (celsius * 9/5) + 32

def fahrenheit_to_celsius(fahrenheit):
    """Convert Fahrenheit to Celsius"""
    return (fahrenheit - 32) * 5/9

def celsius_to_kelvin(celsius):
    """Convert Celsius to Kelvin"""
    return celsius + 273.15
```

Main Program: weather.py

```
# weather.py
import temp_converter

# Test conversions
```

```
temp_c = 25
temp_f = temp_converter.celsius_to_fahrenheit(temp_c)
print(f"{temp_c}°C = {temp_f}°F")

temp_f2 = 77
temp_c2 = temp_converter.fahrenheit_to_celsius(temp_f2)
print(f"{temp_f2}°F = {temp_c2:.2f}°C")

temp_k = temp_converter.celsius_to_kelvin(0)
print(f"0°C = {temp_k}K")
```

Output:

```
25°C = 77.0°F
77°F = 25.00°C
0°C = 273.15K
```

14.3 Variations on the "import" Statement

Python provides several ways to import modules and their contents.

Method 1: Basic import

```
import math_utils

result = math_utils.add(5, 3)
```

Method 2: import with alias

```
import math_utils as mu

result = mu.add(5, 3)
```

Method 3: from...import specific items

```
from math_utils import add, multiply

result1 = add(5, 3)      # No module prefix needed
result2 = multiply(4, 2)
```

Method 4: from...import with alias

```
from math_utils import add as addition

result = addition(10, 20)
```

Method 5: from...import * (import all)

```
from math_utils import *

# All functions available without prefix
result1 = add(5, 3)
result2 = subtract(10, 4)
result3 = multiply(3, 7)
```

⚠ **Warning**: Using `import *` is generally discouraged as it can lead to namespace pollution and makes code harder to debug.

Complete Example

Module: geometry.py

```
# geometry.py
import math

def circle_area(radius):
    """Calculate area of a circle"""
    return math.pi * radius ** 2

def circle_circumference(radius):
    """Calculate circumference of a circle"""
    return 2 * math.pi * radius

def rectangle_area(length, width):
    """Calculate area of a rectangle"""
    return length * width

def rectangle_perimeter(length, width):
    """Calculate perimeter of a rectangle"""
    return 2 * (length + width)
```

Using different import methods:

```
# Method 1: Basic import
import geometry
print(f"Circle area: {geometry.circle_area(5)}")

# Method 2: Import with alias
import geometry as geo
print(f"Rectangle area: {geo.rectangle_area(4, 6)}")

# Method 3: Import specific functions
from geometry import circle_circumference, rectangle_perimeter
print(f"Circle circumference: {circle_circumference(3)}")
print(f"Rectangle perimeter: {rectangle_perimeter(5, 8)}")
```

Output:

```
Circle area: 78.53981633974483
Rectangle area: 24
Circle circumference: 18.84955592153876
Rectangle perimeter: 26
```

14.4 Using the "__all__" Symbol

The `__all__` variable controls what gets imported when using `from module import *`.

Without __all__

```
# my_module.py (without __all__)
def public_func1():
    return "Public 1"

def public_func2():
    return "Public 2"

def _private_func():
    return "Private"

module_var = 42
```

When using `from my_module import *`, all non-underscore names are imported.

With __all__

```
# my_module.py (with __all__)
__all__ = ['public_func1', 'module_var']

def public_func1():
    return "Public 1"

def public_func2():
    return "Public 2"

def _private_func():
    return "Private"

module_var = 42
```

Now, `from my_module import *` only imports `public_func1` and `module_var`.

Complete Example

Module: string_tools.py

```
# string_tools.py
__all__ = ['capitalize_words', 'reverse_string', 'count_vowels']
```

```
def capitalize_words(text):
    """Capitalize first letter of each word"""
    return text.title()

def reverse_string(text):
    """Reverse a string"""
    return text[::-1]

def count_vowels(text):
    """Count vowels in text"""
    vowels = 'aeiouAEIOU'
    return sum(1 for char in text if char in vowels)

def _helper_function():
    """Private helper function"""
    return "This is private"

def not_exported():
    """This won't be imported with *"""
    return "Not in __all__"
```

Usage:

```
# test_string_tools.py
from string_tools import *

# These work (in __all__)
print(capitalize_words("hello world"))
print(reverse_string("Python"))
print(count_vowels("Programming"))

# These won't work (not in __all__)
# print(not_exported())  # Error!
# print(_helper_function())  # Error!
```

Output:

```
Hello World
nohtyP
3
```

14.5 Public and Private Module Variables

Python uses naming conventions to indicate variable and function visibility.

Public Variables

Variables and functions without a leading underscore are considered **public**.

```
# config.py
app_name = "MyApp"           # Public
```

```
version = "1.0.0"            # Public
debug_mode = True            # Public

def get_settings():          # Public
    return f"{app_name} v{version}"
```

Private Variables

Variables and functions starting with a single underscore `_` are considered **private** (by convention).

```
# config.py
_database_password = "secret123"    # Private
_internal_counter = 0               # Private

def _validate_config():             # Private
    pass
```

Complete Example

Module: bank_account.py

```
# bank_account.py
"""Bank account module demonstrating public/private members"""

# Public variables
account_type = "Savings"
min_balance = 100

# Private variables
_transaction_log = []
_account_id = "ACC001"

def deposit(amount):
    """Public function: Deposit money"""
    if amount &gt; 0:
        _update_balance(amount)
        _log_transaction("deposit", amount)
        return f"Deposited ${amount}"
    return "Invalid amount"

def get_balance():
    """Public function: Get current balance"""
    return _current_balance

def _update_balance(amount):
    """Private function: Update balance"""
    global _current_balance
    _current_balance = _current_balance + amount if '_current_balance' in globals() else

def _log_transaction(trans_type, amount):
    """Private function: Log transaction"""
    _transaction_log.append({
        'type': trans_type,
```

```
        'amount': amount
    })

# Initialize
_current_balance = 1000.0
```

Usage:

```
# test_bank.py
import bank_account

# Public access works
print(bank_account.account_type)
print(bank_account.deposit(500))
print(f"Balance: ${bank_account.get_balance()}")

# Private access (possible but discouraged)
print(f"Account ID: {bank_account._account_id}")  # Works but not recommended
```

Output:

```
Savings
Deposited $500
Balance: $1500.0
Account ID: ACC001
```

14.6 The Main Module and "__main__"

The special variable `__name__` helps distinguish between a module being imported vs. being run directly.

Understanding __name__

- When a module is **run directly**: `__name__ == "__main__"`
- When a module is **imported**: `__name__` is the module's name

Example

Module: greetings.py

```
# greetings.py
def say_hello(name):
    """Public function"""
    return f"Hello, {name}!"

def say_goodbye(name):
    """Public function"""
    return f"Goodbye, {name}!"

# Code in this block only runs when module is executed directly
```

```
if __name__ == "__main__":
    print("Running greetings.py directly")
    print(say_hello("Alice"))
    print(say_goodbye("Bob"))
    print(f"Module name: {__name__}")
else:
    print(f"greetings.py imported as {__name__}")
```

Test 1: Run directly

```
$ python greetings.py
```

Output:

```
Running greetings.py directly
Hello, Alice!
Goodbye, Bob!
Module name: __main__
```

Test 2: Import the module

```
# app.py
import greetings

print(greetings.say_hello("Charlie"))
```

Output:

```
greetings.py imported as greetings
Hello, Charlie!
```

Practical Example: Calculator Module

Module: calculator.py

```
# calculator.py
"""Calculator module with test code"""

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    if y == 0:
```

```
        return "Error: Division by zero"
    return x / y

def run_tests():
    """Test all calculator functions"""
    print("Running Calculator Tests...")
    print(f"add(5, 3) = {add(5, 3)}")
    print(f"subtract(10, 4) = {subtract(10, 4)}")
    print(f"multiply(6, 7) = {multiply(6, 7)}")
    print(f"divide(20, 5) = {divide(20, 5)}")
    print(f"divide(10, 0) = {divide(10, 0)}")

# Run tests only when executed directly
if __name__ == "__main__":
    print("Calculator module running as main program")
    run_tests()
```

Output when run directly:

```
Calculator module running as main program
Running Calculator Tests...
add(5, 3) = 8
subtract(10, 4) = 6
multiply(6, 7) = 42
divide(20, 5) = 4.0
divide(10, 0) = Error: Division by zero
```

14.7 Gotcha! Problems with Mutual Importing

Circular imports occur when two modules import each other, potentially causing errors.

Problem Example

Module: module_a.py

```
# module_a.py
import module_b

def func_a():
    print("Function A")
    module_b.func_b()

value_a = 100
```

Module: module_b.py

```
# module_b.py
import module_a

def func_b():
    print("Function B")
```

```
    print(f"Value from A: {module_a.value_a}")

value_b = 200
```

Result: This creates a circular dependency that may cause import errors.

Solution 1: Restructure Code

Move shared functionality to a third module.

Module: common.py

```
# common.py
def shared_function():
    print("Shared functionality")
```

Module: module_a.py

```
# module_a.py
import common

def func_a():
    common.shared_function()
    print("Function A")
```

Module: module_b.py

```
# module_b.py
import common

def func_b():
    common.shared_function()
    print("Function B")
```

Solution 2: Import Inside Function

```
# module_a.py
def func_a():
    import module_b  # Import inside function
    print("Function A")
    module_b.func_b()
```

Solution 3: Avoid Circular Dependencies

Design modules with clear hierarchy:

- High-level modules import low-level modules
- Low-level modules don't import high-level modules

14.8 RPN Example: Breaking into Two Modules

Let's build a **Reverse Polish Notation (RPN) calculator** split across two modules.

Background: What is RPN?

RPN is a mathematical notation where operators follow operands:

- Standard: `3 + 4`
- RPN: `3 4 +`

Module 1: rpn_engine.py

```
# rpn_engine.py
"""RPN Calculator Engine - handles stack operations"""

stack = []

def push(value):
    """Push a value onto the stack"""
    stack.append(float(value))

def pop():
    """Pop and return top value from stack"""
    if len(stack) == 0:
        return None
    return stack.pop()

def peek():
    """Return top value without removing it"""
    if len(stack) == 0:
        return None
    return stack[-1]

def add():
    """Add top two values"""
    if len(stack) &lt; 2:
        return False
    b = pop()
    a = pop()
    push(a + b)
    return True

def subtract():
    """Subtract top value from second value"""
    if len(stack) &lt; 2:
        return False
    b = pop()
    a = pop()
    push(a - b)
    return True

def multiply():
    """Multiply top two values"""
```

```
    if len(stack) &lt; 2:
        return False
    b = pop()
    a = pop()
    push(a * b)
    return True

def divide():
    """Divide second value by top value"""
    if len(stack) &lt; 2:
        return False
    b = pop()
    if b == 0:
        push(b)  # Push back
        return False
    a = pop()
    push(a / b)
    return True

def clear():
    """Clear the stack"""
    global stack
    stack = []

def get_stack():
    """Return current stack"""
    return stack.copy()
```

Module 2: rpn_interp.py

```
# rpn_interp.py
"""RPN Calculator Interpreter - processes commands"""

import rpn_engine

def process(token):
    """Process a single token"""
    token = token.strip()

    if token == '+':
        return rpn_engine.add()
    elif token == '-':
        return rpn_engine.subtract()
    elif token == '*':
        return rpn_engine.multiply()
    elif token == '/':
        return rpn_engine.divide()
    elif token == '.':
        # Print top of stack
        val = rpn_engine.peek()
        if val is not None:
            print(f"Top: {val}")
        else:
            print("Stack empty")
        return True
```

```
    elif token == 'c':
        # Clear stack
        rpn_engine.clear()
        return True
    else:
        # Try to push as number
        try:
            rpn_engine.push(float(token))
            return True
        except ValueError:
            return False

def run(program):
    """Run an RPN program (list of tokens)"""
    for token in program:
        success = process(token)
        if not success:
            print(f"Error processing: {token}")
            return False
    return True

# Test code
if __name__ == "__main__":
    print("RPN Calculator Test")

    # Test: 3 4 + (should be 7)
    program1 = ['3', '4', '+', '.']
    print("Program: 3 4 +")
    run(program1)

    # Test: 10 5 - 2 * (should be 10)
    rpn_engine.clear()
    program2 = ['10', '5', '-', '2', '*', '.']
    print("\nProgram: 10 5 - 2 *")
    run(program2)

    # Test: 6 3 / 4 + (should be 6)
    rpn_engine.clear()
    program3 = ['6', '3', '/', '4', '+', '.']
    print("\nProgram: 6 3 / 4 +")
    run(program3)
```

Output:

```
RPN Calculator Test
Program: 3 4 +
Top: 7.0

Program: 10 5 - 2 *
Top: 10.0

Program: 6 3 / 4 +
Top: 6.0
```

14.9 RPN Example: Adding I/O Directives

Let's enhance our RPN calculator with input/output directives.

Enhanced rpn_interp.py

```
# rpn_interp.py (enhanced)
"""RPN Calculator with I/O directives"""

import rpn_engine

def process(token):
    """Process a single token with I/O support"""
    token = token.strip()

    # Arithmetic operators
    if token == '+':
        return rpn_engine.add()
    elif token == '-':
        return rpn_engine.subtract()
    elif token == '*':
        return rpn_engine.multiply()
    elif token == '/':
        return rpn_engine.divide()

    # I/O directives
    elif token == '.':
        # Print top of stack
        val = rpn_engine.peek()
        if val is not None:
            print(f"{val}")
        else:
            print("Stack empty")
        return True

    elif token == '.s':
        # Show entire stack
        stack = rpn_engine.get_stack()
        if len(stack) == 0:
            print("Stack: []")
        else:
            print(f"Stack: {stack}")
        return True

    elif token == 'c':
        # Clear stack
        rpn_engine.clear()
        print("Stack cleared")
        return True

    elif token == 'p':
        # Print and pop
        val = rpn_engine.pop()
        if val is not None:
            print(f"{val}")
```

```
            else:
                print("Stack empty")
            return True

        else:
            # Try to push as number
            try:
                rpn_engine.push(float(token))
                return True
            except ValueError:
                print(f"Unknown token: {token}")
                return False

    def run(program):
        """Run an RPN program"""
        for token in program:
            success = process(token)
            if not success:
                return False
        return True

    def run_interactive():
        """Run calculator in interactive mode"""
        print("RPN Calculator (type 'quit' to exit)")
        print("Commands: +, -, *, /, ., .s, p, c")

        while True:
            try:
                line = input("RPN&gt; ").strip()
                if line.lower() == 'quit':
                    break
                if line:
                    tokens = line.split()
                    run(tokens)
            except EOFError:
                break

    # Test code
    if __name__ == "__main__":
        # Test I/O directives
        print("=== Test 1: Basic operations with print ===")
        program1 = ['5', '3', '+', '.', '2', '*', '.']
        run(program1)

        print("\n=== Test 2: Show stack ===")
        rpn_engine.clear()
        program2 = ['10', '20', '30', '.s']
        run(program2)

        print("\n=== Test 3: Print and pop ===")
        rpn_engine.clear()
        program3 = ['7', '8', '9', 'p', 'p', 'p']
        run(program3)

        print("\n=== Test 4: Clear stack ===")
        rpn_engine.clear()
```

```
    program4 = ['1', '2', '3', '.s', 'c', '.s']
    run(program4)
```

Output:

```
=== Test 1: Basic operations with print ===
8.0
16.0

=== Test 2: Show stack ===
Stack: [10.0, 20.0, 30.0]

=== Test 3: Print and pop ===
9.0
8.0
7.0

=== Test 4: Clear stack ===
Stack: [1.0, 2.0, 3.0]
Stack cleared
Stack: []
```

14.10 Further Changes to the RPN Example

Let's add more advanced features to our RPN calculator.

14.10.1 Adding Line-Number Checking

Enhanced rpn_engine.py with line tracking:

```
# rpn_engine.py (with line tracking)
"""RPN Engine with line number tracking"""

stack = []
current_line = 0

def push(value):
    """Push a value onto the stack"""
    stack.append(float(value))

def pop():
    """Pop and return top value from stack"""
    if len(stack) == 0:
        print(f"Error at line {current_line}: Stack underflow")
        return None
    return stack.pop()

def add():
    """Add top two values"""
    if len(stack) &lt; 2:
        print(f"Error at line {current_line}: Need 2 values for +")
        return False
    b = pop()
```

```
    a = pop()
    if a is not None and b is not None:
        push(a + b)
        return True
    return False

def subtract():
    """Subtract top value from second value"""
    if len(stack) &lt; 2:
        print(f"Error at line {current_line}: Need 2 values for -")
        return False
    b = pop()
    a = pop()
    if a is not None and b is not None:
        push(a - b)
        return True
    return False

def multiply():
    """Multiply top two values"""
    if len(stack) &lt; 2:
        print(f"Error at line {current_line}: Need 2 values for *")
        return False
    b = pop()
    a = pop()
    if a is not None and b is not None:
        push(a * b)
        return True
    return False

def divide():
    """Divide second value by top value"""
    if len(stack) &lt; 2:
        print(f"Error at line {current_line}: Need 2 values for /")
        return False
    b = pop()
    if b == 0:
        print(f"Error at line {current_line}: Division by zero")
        push(b)
        return False
    a = pop()
    if a is not None and b is not None:
        push(a / b)
        return True
    return False

def set_line_number(line):
    """Set current line number"""
    global current_line
    current_line = line

def get_stack():
    """Return current stack"""
    return stack.copy()

def clear():
```

```
    """Clear the stack"""
    global stack
    stack = []
```

Test with line numbers:

```
# test_line_numbers.py
import rpn_engine
import rpn_interp

# Test error reporting
print("=== Test: Stack underflow ===")
rpn_engine.clear()
rpn_engine.set_line_number(1)
rpn_interp.process('5')
rpn_engine.set_line_number(2)
rpn_interp.process('+')  # Error: need 2 values

print("\n=== Test: Division by zero ===")
rpn_engine.clear()
rpn_engine.set_line_number(1)
rpn_interp.process('10')
rpn_engine.set_line_number(2)
rpn_interp.process('0')
rpn_engine.set_line_number(3)
rpn_interp.process('/')  # Error: division by zero
```

Output:

```
=== Test: Stack underflow ===
Error at line 2: Need 2 values for +

=== Test: Division by zero ===
Error at line 3: Division by zero
```

14.10.2 Adding Jump-If-Not-Zero

Let's add conditional execution capability.

Enhanced rpn_interp.py with jump:

```
# rpn_interp.py (with jump support)
"""RPN Interpreter with jump support"""

import rpn_engine

program_counter = 0
labels = {}

def parse_program(program):
    """Parse program and extract labels"""
    global labels
```

```
    labels = {}
    for i, token in enumerate(program):
        if token.startswith(':'):
            label_name = token[1:]
            labels[label_name] = i
    return program

def process(token):
    """Process a single token with jump support"""
    token = token.strip()

    # Skip label definitions
    if token.startswith(':'):
        return True

    # Jump if not zero
    if token.startswith('?'):
        label = token[1:]
        val = rpn_engine.pop()
        if val is not None and val != 0:
            if label in labels:
                global program_counter
                program_counter = labels[label]
                return True
            else:
                print(f"Error: Label '{label}' not found")
                return False
        return True

    # Arithmetic operators
    if token == '+':
        return rpn_engine.add()
    elif token == '-':
        return rpn_engine.subtract()
    elif token == '*':
        return rpn_engine.multiply()
    elif token == '/':
        return rpn_engine.divide()
    elif token == '.':
        val = rpn_engine.peek()
        if val is not None:
            print(f"{val}")
        return True
    elif token == 'c':
        rpn_engine.clear()
        return True
    else:
        try:
            rpn_engine.push(float(token))
            return True
        except ValueError:
            return False

def run(program):
    """Run program with jump support"""
    global program_counter
```

```
    program = parse_program(program)
    program_counter = 0

    while program_counter &lt; len(program):
        token = program[program_counter]
        rpn_engine.set_line_number(program_counter + 1)
        success = process(token)
        if not success:
            return False
        program_counter += 1
    return True

# Test
if __name__ == "__main__":
    print("=== Test: Simple loop with jump ===")
    # Count down from 5
    program = [
        '5',           # Push 5
        ':loop',       # Label
        '.',           # Print
        '1',           # Push 1
        '-',           # Subtract
        '?loop'        # Jump to loop if not zero
    ]

    rpn_engine.clear()
    run(program)
```

Output:

```
=== Test: Simple loop with jump ===
5.0
4.0
3.0
2.0
1.0
```

14.10.3 Greater-Than (>) and Get-Random-Number (!)

Final enhancements to rpn_engine.py:

```
# rpn_engine.py (final version)
"""RPN Engine with comparison and random"""

import random

stack = []
current_line = 0

def push(value):
    """Push a value onto the stack"""
    stack.append(float(value))
```

```
def pop():
    """Pop and return top value from stack"""
    if len(stack) == 0:
        print(f"Error at line {current_line}: Stack underflow")
        return None
    return stack.pop()

def add():
    """Add top two values"""
    if len(stack) &lt; 2:
        return False
    b = pop()
    a = pop()
    if a is not None and b is not None:
        push(a + b)
        return True
    return False

def subtract():
    """Subtract top value from second value"""
    if len(stack) &lt; 2:
        return False
    b = pop()
    a = pop()
    if a is not None and b is not None:
        push(a - b)
        return True
    return False

def multiply():
    """Multiply top two values"""
    if len(stack) &lt; 2:
        return False
    b = pop()
    a = pop()
    if a is not None and b is not None:
        push(a * b)
        return True
    return False

def divide():
    """Divide second value by top value"""
    if len(stack) &lt; 2:
        return False
    b = pop()
    if b == 0:
        print(f"Error at line {current_line}: Division by zero")
        push(b)
        return False
    a = pop()
    if a is not None and b is not None:
        push(a / b)
        return True
    return False

def greater_than():
```

```
    """Compare: push 1 if second &gt; top, else 0"""
    if len(stack) &lt; 2:
        return False
    b = pop()
    a = pop()
    if a is not None and b is not None:
        push(1.0 if a &gt; b else 0.0)
        return True
    return False

def random_number():
    """Push random number between 0 and 1"""
    push(random.random())
    return True

def set_line_number(line):
    """Set current line number"""
    global current_line
    current_line = line

def get_stack():
    """Return current stack"""
    return stack.copy()

def clear():
    """Clear the stack"""
    global stack
    stack = []
```

Enhanced rpn_interp.py:

```
# rpn_interp.py (final version)
"""RPN Interpreter - complete version"""

import rpn_engine

program_counter = 0
labels = {}

def parse_program(program):
    """Parse program and extract labels"""
    global labels
    labels = {}
    for i, token in enumerate(program):
        if token.startswith(':'):
            label_name = token[1:]
            labels[label_name] = i
    return program

def process(token):
    """Process a single token - complete version"""
    token = token.strip()

    # Skip label definitions
    if token.startswith(':'):
```

```
        return True

    # Jump if not zero
    if token.startswith('?'):
        label = token[1:]
        val = rpn_engine.pop()
        if val is not None and val != 0:
            if label in labels:
                global program_counter
                program_counter = labels[label]
                return True
        return True

    # Arithmetic operators
    if token == '+':
        return rpn_engine.add()
    elif token == '-':
        return rpn_engine.subtract()
    elif token == '*':
        return rpn_engine.multiply()
    elif token == '/':
        return rpn_engine.divide()
    elif token == '&gt;':
        return rpn_engine.greater_than()
    elif token == '!':
        return rpn_engine.random_number()
    elif token == '.':
        val = rpn_engine.peek()
        if val is not None:
            print(f"{val}")
        return True
    elif token == '.s':
        print(f"Stack: {rpn_engine.get_stack()}")
        return True
    elif token == 'c':
        rpn_engine.clear()
        return True
    elif token == 'p':
        val = rpn_engine.pop()
        if val is not None:
            print(f"{val}")
        return True
    else:
        try:
            rpn_engine.push(float(token))
            return True
        except ValueError:
            print(f"Unknown token: {token}")
            return False

def run(program):
    """Run program"""
    global program_counter
    program = parse_program(program)
    program_counter = 0
```

```
    while program_counter &lt; len(program):
        token = program[program_counter]
        rpn_engine.set_line_number(program_counter + 1)
        success = process(token)
        if not success:
            return False
        program_counter += 1
    return True

# Tests
if __name__ == "__main__":
    print("=== Test 1: Greater than ===")
    rpn_engine.clear()
    program1 = ['10', '5', '&gt;', '.']  # Should print 1.0
    run(program1)

    rpn_engine.clear()
    program2 = ['3', '8', '&gt;', '.']   # Should print 0.0
    run(program2)

    print("\n=== Test 2: Random number ===")
    rpn_engine.clear()
    program3 = ['!', '.', '!', '.', '!', '.']
    run(program3)

    print("\n=== Test 3: Combined example ===")
    rpn_engine.clear()
    program4 = ['!', '100', '*', '.']  # Random * 100
    run(program4)
```

Output:

```
=== Test 1: Greater than ===
1.0
0.0

=== Test 2: Random number ===
0.8472135954999579
0.3958453154293345
0.6623847921913842

=== Test 3: Combined example ===
73.19939418114051
```

14.11 RPN: Putting It All Together

Let's create a complete, production-ready RPN calculator.

Complete rpn_engine.py

```
# rpn_engine.py (complete)
"""
RPN Calculator Engine - Complete Version
Handles all stack operations and arithmetic
"""

import random

# Global state
stack = []
current_line = 0

def push(value):
    """Push a value onto the stack"""
    try:
        stack.append(float(value))
        return True
    except:
        return False

def pop():
    """Pop and return top value from stack"""
    if len(stack) == 0:
        return None
    return stack.pop()

def peek():
    """Return top value without removing it"""
    if len(stack) == 0:
        return None
    return stack[-1]

def add():
    """Add top two values"""
    if len(stack) &lt; 2:
        return False
    b, a = pop(), pop()
    if a is not None and b is not None:
        push(a + b)
        return True
    return False

def subtract():
    """Subtract top value from second value"""
    if len(stack) &lt; 2:
        return False
    b, a = pop(), pop()
    if a is not None and b is not None:
        push(a - b)
        return True
    return False

def multiply():
    """Multiply top two values"""
```

```
    if len(stack) &lt; 2:
        return False
    b, a = pop(), pop()
    if a is not None and b is not None:
        push(a * b)
        return True
    return False

def divide():
    """Divide second value by top value"""
    if len(stack) &lt; 2:
        return False
    b = pop()
    if b == 0:
        push(b)
        return False
    a = pop()
    if a is not None and b is not None:
        push(a / b)
        return True
    return False

def greater_than():
    """Compare: push 1 if second &gt; top, else 0"""
    if len(stack) &lt; 2:
        return False
    b, a = pop(), pop()
    if a is not None and b is not None:
        push(1.0 if a &gt; b else 0.0)
        return True
    return False

def random_number():
    """Push random number between 0 and 1"""
    push(random.random())
    return True

def duplicate():
    """Duplicate top of stack"""
    val = peek()
    if val is not None:
        push(val)
        return True
    return False

def swap():
    """Swap top two values"""
    if len(stack) &lt; 2:
        return False
    b, a = pop(), pop()
    push(b)
    push(a)
    return True

def set_line_number(line):
    """Set current line number for error reporting"""
```

```
    global current_line
    current_line = line

def get_line_number():
    """Get current line number"""
    return current_line

def get_stack():
    """Return copy of current stack"""
    return stack.copy()

def clear():
    """Clear the stack"""
    global stack
    stack = []

def size():
    """Return stack size"""
    return len(stack)
```

Complete rpn_interp.py

```
# rpn_interp.py (complete)
"""
RPN Calculator Interpreter - Complete Version
Processes tokens and manages program execution
"""

import rpn_engine

# Global state
program_counter = 0
labels = {}
program = []

def parse_program(prog):
    """Parse program and extract labels"""
    global labels, program
    labels = {}
    program = prog

    for i, token in enumerate(program):
        if token.startswith(':'):
            label_name = token[1:]
            labels[label_name] = i

    return program

def process(token):
    """Process a single token"""
    global program_counter

    token = token.strip()

    # Skip label definitions
```

```
    if token.startswith(':'):
        return True

    # Jump if not zero
    if token.startswith('?'):
        label = token[1:]
        val = rpn_engine.pop()
        if val is not None and val != 0:
            if label in labels:
                program_counter = labels[label] - 1
                return True
            else:
                print(f"Error: Undefined label '{label}'")
                return False
        return True

    # Arithmetic operators
    if token == '+':
        return rpn_engine.add()
    elif token == '-':
        return rpn_engine.subtract()
    elif token == '*':
        return rpn_engine.multiply()
    elif token == '/':
        if not rpn_engine.divide():
            print("Error: Division failed")
            return False
        return True

    # Comparison
    elif token == '&gt;':
        return rpn_engine.greater_than()

    # Random
    elif token == '!':
        return rpn_engine.random_number()

    # Stack operations
    elif token == 'dup':
        return rpn_engine.duplicate()
    elif token == 'swap':
        return rpn_engine.swap()

    # I/O operations
    elif token == '.':
        val = rpn_engine.peek()
        if val is not None:
            print(f"{val}")
        else:
            print("Stack empty")
        return True

    elif token == '.s':
        stack = rpn_engine.get_stack()
        if len(stack) == 0:
            print("Stack: []")
```

```
        else:
            print(f"Stack: {stack}")
        return True

    elif token == 'p':
        val = rpn_engine.pop()
        if val is not None:
            print(f"{val}")
        else:
            print("Stack empty")
        return True

    # Control
    elif token == 'c':
        rpn_engine.clear()
        return True

    elif token == 'size':
        print(f"Stack size: {rpn_engine.size()}")
        return True

    # Numbers
    else:
        try:
            rpn_engine.push(float(token))
            return True
        except ValueError:
            print(f"Error: Unknown token '{token}'")
            return False

def run(prog):
    """Run an RPN program"""
    global program_counter

    parse_program(prog)
    program_counter = 0

    while program_counter &lt; len(program):
        token = program[program_counter]
        rpn_engine.set_line_number(program_counter + 1)

        success = process(token)
        if not success:
            return False

        program_counter += 1

    return True

def run_file(filename):
    """Run program from file"""
    try:
        with open(filename, 'r') as f:
            tokens = []
            for line in f:
                line = line.split('#')[0]  # Remove comments
```

```
                tokens.extend(line.split())
            return run(tokens)
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found")
        return False

def run_interactive():
    """Run in interactive mode"""
    print("RPN Calculator - Interactive Mode")
    print("Type 'help' for commands, 'quit' to exit")

    while True:
        try:
            line = input("RPN&gt; ").strip()

            if line.lower() == 'quit':
                print("Goodbye!")
                break

            if line.lower() == 'help':
                print_help()
                continue

            if line:
                tokens = line.split()
                run(tokens)

        except EOFError:
            break
        except KeyboardInterrupt:
            print("\nInterrupted")
            break

def print_help():
    """Print help information"""
    help_text = """
RPN Calculator Commands:
------------------------
Numbers: Push onto stack (e.g., 3.14, 42, -7)
Operators:
  +       Add top two values
  -       Subtract top from second
  *       Multiply top two values
  /       Divide second by top
  &gt;       Compare (1 if second &gt; top, else 0)

Stack Operations:
  dup     Duplicate top of stack
  swap    Swap top two values
  c       Clear stack

I/O:
  .       Print top of stack
  .s      Show entire stack
  p       Print and pop
  size    Show stack size
```

```
Control:
  :label  Define label
  ?label  Jump to label if top != 0

Special:
  !       Push random number (0-1)
  help    Show this help
  quit    Exit program
"""
    print(help_text)

# Main execution
if __name__ == "__main__":
    print("=== RPN Calculator - Complete Demo ===\n")

    # Demo 1: Basic arithmetic
    print("Demo 1: Basic Arithmetic")
    print("Program: 5 3 + 2 *")
    rpn_engine.clear()
    run(['5', '3', '+', '2', '*', '.'])

    # Demo 2: Complex expression
    print("\nDemo 2: Complex Expression")
    print("Program: 10 5 3 + * 2 /")
    rpn_engine.clear()
    run(['10', '5', '3', '+', '*', '2', '/', '.'])

    # Demo 3: Comparison
    print("\nDemo 3: Comparison")
    print("Program: 8 5 &gt; .")
    rpn_engine.clear()
    run(['8', '5', '&gt;', '.'])

    # Demo 4: Stack operations
    print("\nDemo 4: Stack Operations")
    print("Program: 7 dup * .")
    rpn_engine.clear()
    run(['7', 'dup', '*', '.'])

    # Demo 5: Loop (countdown)
    print("\nDemo 5: Countdown Loop")
    print("Program: 5 :loop . 1 - dup ?loop")
    rpn_engine.clear()
    run(['5', ':loop', '.', '1', '-', 'dup', '?loop'])

    print("\n=== Demo Complete ===")
```

Output:

```
=== RPN Calculator - Complete Demo ===

Demo 1: Basic Arithmetic
Program: 5 3 + 2 *
16.0
```

```
Demo 2: Complex Expression
Program: 10 5 3 + * 2 /
40.0

Demo 3: Comparison
Program: 8 5 &gt; .
1.0

Demo 4: Stack Operations
Program: 7 dup * .
49.0

Demo 5: Countdown Loop
Program: 5 :loop . 1 - dup ?loop
5.0
4.0
3.0
2.0
1.0

=== Demo Complete ===
```

Chapter Summary

Key Concepts Covered

1. **Modules**: Files containing Python code that can be imported and reused
2. **Import Statements**: Various ways to import modules (`import`, `from...import`, aliases)
3. **__all__**: Control what gets exported with `from module import *`
4. **Public/Private**: Naming conventions for module visibility (underscore prefix)
5. **__name__ and __main__**: Execute code only when module runs directly
6. **Circular Imports**: Problems with mutual importing and solutions
7. **Multi-Module Projects**: Organizing large programs across multiple files

Best Practices

✅ **Do:**

- Use descriptive module names
- Document modules with docstrings
- Keep modules focused on single responsibility
- Use `__name__ == "__main__"` for test code
- Follow naming conventions for visibility

❌ **Don't:**

- Use `from module import *` in production code

- Create circular dependencies
- Mix unrelated functionality in one module
- Forget to document public interfaces

RPN Calculator Features

The complete RPN calculator demonstrates:

- **Modular design**: Separated engine and interpreter
- **Stack operations**: push, pop, peek
- **Arithmetic**: +, -, *, /
- **Comparisons**: >
- **Control flow**: Labels and conditional jumps
- **I/O**: Multiple print and display options
- **Error handling**: Line numbers, stack underflow, division by zero
- **Random numbers**: Built-in random generation

Practice Exercises

Exercise 1: Statistics Module

Create a `stats.py` module with functions for mean, median, and standard deviation.

Exercise 2: File Utilities

Build a `file_utils.py` module with functions for reading, writing, and processing text files.

Exercise 3: Shopping Cart

Design a multi-module shopping cart system with separate modules for products, cart, and checkout.

Exercise 4: RPN Extensions

Extend the RPN calculator with:

- Trigonometric functions (sin, cos, tan)
- Exponentiation operator
- Save/load stack to file
- Command history

Conclusion

Modular programming is essential for building maintainable, scalable Python applications. By organizing code into modules, you create reusable components that can be tested independently and combined to build complex systems.

The RPN calculator example demonstrates how to design a multi-module application with clear separation of concerns: the engine handles core operations while the interpreter manages user interaction and control flow.

Key Takeaways:

- Modules promote code organization and reusability
- Import mechanisms provide flexibility in accessing module contents
- Naming conventions guide visibility and usage
- The `__name__` variable enables dual-purpose modules
- Proper design avoids circular dependency problems

End of Chapter 14 Tutorial