Tutorial by Dr. Ayes Chinmay
Date: 16/10/2025

Chapter 8: Text and Binary Files - Complete Tutorial

Table of Contents

8.1 Two Kinds of Files: Text and Binary

Python can work with two fundamental types of files: **text files** and **binary files**. Understanding the difference between these two types is crucial for effective file handling.

8.1.1 Text Files

Text files contain human-readable characters and are encoded using character encodings like UTF-8, ASCII, or Unicode. When Python reads a text file, it automatically converts the bytes to strings.

Key characteristics of text files:

- Contain readable text characters
- Use character encodings (UTF-8, ASCII, etc.)
- Automatically handle encoding/decoding
- Platform-specific line endings (\n, \r\n, \r)

Example 1: Basic Text File Writing

```
# Writing to a text file
def write_text_file():
    # Open file in write mode
    file = open('sample.txt', 'w')
    file.write('Hello, World!\n')
    file.write('This is a text file.\n')
    file.write('Python file handling is easy!')
    file.close()
    print("Text file created successfully!")

# Execute the function
write_text_file()
```

Output:

```
Text file created successfully!
```

File content (sample.txt):

```
Hello, World!
This is a text file.
Python file handling is easy!
```

Example 2: Basic Text File Reading

```
# Reading from a text file
def read_text_file():
    file = open('sample.txt', 'r')
    content = file.read()
    file.close()
```

```
    return content

# Read and display the file content
content = read_text_file()
print("File content:")
print(content)
```

Output:

```
File content:
Hello, World!
This is a text file.
Python file handling is easy!
```

Example 3: Reading Line by Line

```
# Reading a text file line by line
def read_lines():
    file = open('sample.txt', 'r')
    print("Reading line by line:")
    line_number = 1
    for line in file:
        print(f"Line {line_number}: {line.strip()}")
        line_number += 1
    file.close()

read_lines()
```

Output:

```
Reading line by line:
Line 1: Hello, World!
Line 2: This is a text file.
Line 3: Python file handling is easy!
```

8.1.2 Binary Files

Binary files contain data in binary format (sequences of bytes) and are not human-readable. They can store any type of data including images, videos, executables, and serialized objects.

Key characteristics of binary files:

- Contain raw bytes
- No automatic encoding/decoding
- Platform-independent
- Can store any type of data

Example 4: Basic Binary File Operations

```
# Writing to a binary file
def write_binary_file():
    # Create some binary data
    data = b'\x48\x65\x6C\x6C\x6F'  # "Hello" in bytes

    file = open('sample.bin', 'wb')
    file.write(data)
    file.close()
    print("Binary file created successfully!")

# Reading from a binary file
def read_binary_file():
    file = open('sample.bin', 'rb')
    data = file.read()
    file.close()
    return data

# Execute the functions
write_binary_file()
binary_data = read_binary_file()
print(f"Binary data: {binary_data}")
print(f"Decoded: {binary_data.decode('utf-8')}")
```

Output:

```
Binary file created successfully!
Binary data: b'Hello'
Decoded: Hello
```

8.2 Approaches to Binary Files: A Summary

There are several approaches to working with binary files in Python:

1. **Direct byte operations** - Reading and writing raw bytes
2. **Struct module** - Converting between Python objects and C structs
3. **Pickle module** - Serializing Python objects
4. **Specialized libraries** - For specific file formats (images, audio, etc.)

Example 5: Different Binary Approaches Demonstration

```
import struct
import pickle

# Approach 1: Direct byte operations
def direct_bytes_example():
    # Writing integers as bytes
    numbers = [1, 2, 3, 4, 5]
    with open('numbers.bin', 'wb') as f:
        for num in numbers:
            f.write(num.to_bytes(4, 'little'))
```

```
    # Reading integers from bytes
    with open('numbers.bin', 'rb') as f:
        result = []
        while True:
            data = f.read(4)
            if not data:
                break
            result.append(int.from_bytes(data, 'little'))

    print(f"Direct bytes - Original: {numbers}")
    print(f"Direct bytes - Read back: {result}")

# Approach 2: Using struct module
def struct_example():
    numbers = [1, 2, 3, 4, 5]

    # Pack integers into binary format
    binary_data = struct.pack('5i', *numbers)

    with open('struct_numbers.bin', 'wb') as f:
        f.write(binary_data)

    # Unpack binary data back to integers
    with open('struct_numbers.bin', 'rb') as f:
        data = f.read()
        result = struct.unpack('5i', data)

    print(f"Struct - Original: {numbers}")
    print(f"Struct - Read back: {list(result)}")

# Execute examples
direct_bytes_example()
struct_example()
```

Output:

```
Direct bytes - Original: [1, 2, 3, 4, 5]
Direct bytes - Read back: [1, 2, 3, 4, 5]
Struct - Original: [1, 2, 3, 4, 5]
Struct - Read back: [1, 2, 3, 4, 5]
```

8.3 The File/Directory System

Understanding file paths and directory operations is essential for effective file handling.

Example 6: Working with File Paths

```
import os

def explore_filesystem():
    # Get current working directory
    current_dir = os.getcwd()
    print(f"Current directory: {current_dir}")
```

```
    # List files in current directory
    files = os.listdir('.')
    print(f"Files in current directory: {files}")

    # Create a new directory
    new_dir = 'test_directory'
    if not os.path.exists(new_dir):
        os.makedirs(new_dir)
        print(f"Created directory: {new_dir}")

    # Create a file in the new directory
    file_path = os.path.join(new_dir, 'test_file.txt')
    with open(file_path, 'w') as f:
        f.write('This is a test file in a subdirectory.')

    print(f"Created file: {file_path}")

    # Check if file exists
    if os.path.exists(file_path):
        print(f"File {file_path} exists!")

        # Get file information
        file_size = os.path.getsize(file_path)
        print(f"File size: {file_size} bytes")

explore_filesystem()
```

Output:

```
Current directory: /path/to/current/directory
Files in current directory: ['sample.txt', 'sample.bin', 'numbers.bin', 'struct_numbers.b
Created directory: test_directory
Created file: test_directory/test_file.txt
File test_directory/test_file.txt exists!
File size: 42 bytes
```

8.4 Handling File-Opening Exceptions

File operations can fail for various reasons. It's important to handle these exceptions gracefully.

Example 7: Exception Handling for File Operations

```
def handle_file_exceptions():
    # Example 1: FileNotFoundError
    try:
        with open('nonexistent_file.txt', 'r') as f:
            content = f.read()
    except FileNotFoundError:
        print("Error: File not found!")

    # Example 2: PermissionError simulation
    try:
```

```
        # Try to write to a read-only file (if it exists)
        with open('readonly_file.txt', 'w') as f:
            f.write("This might fail")
    except PermissionError:
        print("Error: Permission denied!")
    except FileNotFoundError:
        print("Read-only file doesn't exist, creating a normal file instead")
        with open('readonly_file.txt', 'w') as f:
            f.write("Normal file created")

    # Example 3: General exception handling
    filename = 'test_exceptions.txt'
    try:
        with open(filename, 'r') as f:
            content = f.read()
        print(f"Successfully read: {content}")
    except FileNotFoundError:
        print(f"File {filename} not found. Creating it...")
        try:
            with open(filename, 'w') as f:
                f.write("This file was created due to an exception.")
            print("File created successfully!")
        except IOError as e:
            print(f"Could not create file: {e}")
    except IOError as e:
        print(f"IO Error occurred: {e}")

handle_file_exceptions()
```

Output:

```
Error: File not found!
Read-only file doesn't exist, creating a normal file instead
File test_exceptions.txt not found. Creating it...
File created successfully!
```

8.5 Using the 'with' Keyword

The `with` statement provides a context manager that automatically handles file closing, even if an exception occurs.

Example 8: Comparing Traditional vs. 'with' Statement

```
def traditional_file_handling():
    # Traditional way (not recommended)
    try:
        f = open('traditional.txt', 'w')
        f.write('This is the traditional way.')
        f.close()  # Must remember to close
        print("Traditional file handling completed")
    except Exception as e:
        print(f"Error: {e}")
```

```
def with_statement_handling():
    # Using 'with' statement (recommended)
    try:
        with open('with_statement.txt', 'w') as f:
            f.write('This uses the with statement.')
        # File is automatically closed here
        print("With statement file handling completed")
    except Exception as e:
        print(f"Error: {e}")

# Demonstrate both methods
traditional_file_handling()
with_statement_handling()

# Verify files were created and read them
def read_both_files():
    files = ['traditional.txt', 'with_statement.txt']
    for filename in files:
        try:
            with open(filename, 'r') as f:
                content = f.read()
                print(f"{filename}: {content}")
        except FileNotFoundError:
            print(f"{filename} not found")

read_both_files()
```

Output:

```
Traditional file handling completed
With statement file handling completed
traditional.txt: This is the traditional way.
with_statement.txt: This uses the with statement.
```

Example 9: Advanced 'with' Statement Usage

```
def advanced_with_examples():
    # Multiple file operations with 'with'
    with open('input.txt', 'w') as input_file, \
         open('output.txt', 'w') as output_file:

        input_file.write('Line 1\nLine 2\nLine 3\n')
        output_file.write('Processed data\n')

    print("Multiple files handled successfully")

    # Reading and processing
    try:
        with open('input.txt', 'r') as f:
            lines = f.readlines()
            print(f"Read {len(lines)} lines:")
            for i, line in enumerate(lines, 1):
                print(f"  {i}: {line.strip()}")
    except FileNotFoundError:
```

```
        print("Input file not found")

advanced_with_examples()
```

Output:

```
Multiple files handled successfully
Read 3 lines:
  1: Line 1
  2: Line 2
  3: Line 3
```

8.6 Summary of Read/Write Operations

Here's a comprehensive summary of different file operation modes and methods:

Example 10: Complete File Operations Summary

```
def file_operations_summary():
    filename = 'operations_demo.txt'

    # Different file modes demonstration
    modes = {
        'w': 'Write mode (overwrites existing file)',
        'a': 'Append mode (adds to existing file)',
        'r': 'Read mode (read existing file)',
        'x': 'Exclusive creation (fails if file exists)',
        'w+': 'Write and read mode',
        'r+': 'Read and write mode'
    }

    print("File Operation Modes:")
    for mode, description in modes.items():
        print(f"  {mode}: {description}")

    # Demonstrate write mode
    with open(filename, 'w') as f:
        f.write('Line 1 (write mode)\n')
        f.write('Line 2 (write mode)\n')

    print(f"\nCreated {filename} with write mode")

    # Demonstrate append mode
    with open(filename, 'a') as f:
        f.write('Line 3 (append mode)\n')
        f.write('Line 4 (append mode)\n')

    print("Added lines with append mode")

    # Demonstrate different read methods
    with open(filename, 'r') as f:
        print("\nDifferent read methods:")
```

```
        # Reset file pointer to beginning
        f.seek(0)
        all_content = f.read()
        print(f"read(): {repr(all_content)}")

        # Reset and read line by line
        f.seek(0)
        first_line = f.readline()
        print(f"readline(): {repr(first_line)}")

        # Reset and read all lines
        f.seek(0)
        all_lines = f.readlines()
        print(f"readlines(): {all_lines}")

file_operations_summary()
```

Output:

```
File Operation Modes:
  w: Write mode (overwrites existing file)
  a: Append mode (adds to existing file)
  r: Read mode (read existing file)
  x: Exclusive creation (fails if file exists)
  w+: Write and read mode
  r+: Read and write mode

Created operations_demo.txt with write mode
Added lines with append mode

Different read methods:
read(): 'Line 1 (write mode)\nLine 2 (write mode)\nLine 3 (append mode)\nLine 4 (append m
readline(): 'Line 1 (write mode)\n'
readlines(): ['Line 1 (write mode)\n', 'Line 2 (write mode)\n', 'Line 3 (append mode)\n',
```

8.7 Text File Operations in Depth

Text file operations offer various methods for reading and writing data efficiently.

Example 11: Advanced Text File Operations

```
def advanced_text_operations():
    # Create a sample data file
    data_file = 'student_data.txt'

    students = [
        "John Smith,85,92,78",
        "Jane Doe,91,88,95",
        "Bob Johnson,76,82,88",
        "Alice Brown,94,96,91"
    ]

    # Write student data
```

```
    with open(data_file, 'w') as f:
        f.write("Name,Math,Science,English\n")  # Header
        for student in students:
            f.write(student + "\n")

    print(f"Created {data_file} with student data")

    # Read and process the data
    with open(data_file, 'r') as f:
        header = f.readline().strip().split(',')
        print(f"Header: {header}")

        print("\nStudent Records:")
        for line_num, line in enumerate(f, 2):  # Start from line 2
            parts = line.strip().split(',')
            name = parts[^0]
            scores = [int(score) for score in parts[1:]]
            average = sum(scores) / len(scores)

            print(f"Line {line_num}: {name} - Average: {average:.1f}")

advanced_text_operations()
```

Output:

```
Created student_data.txt with student data
Header: ['Name', 'Math', 'Science', 'English']

Student Records:
Line 2: John Smith - Average: 85.0
Line 3: Jane Doe - Average: 91.3
Line 4: Bob Johnson - Average: 82.0
Line 5: Alice Brown - Average: 93.7
```

Example 12: Text File Encoding and Unicode

```
def encoding_examples():
    # Writing with different encodings
    text_data = "Hello, 世界! Café résumé naïve"

    encodings = ['utf-8', 'ascii', 'latin-1']

    for encoding in encodings:
        filename = f'text_{encoding.replace("-", "_")}.txt'
        try:
            with open(filename, 'w', encoding=encoding) as f:
                f.write(text_data)
            print(f"Successfully wrote with {encoding} encoding")

            # Read it back
            with open(filename, 'r', encoding=encoding) as f:
                content = f.read()
            print(f"Read back: {content}")
```

```
        except UnicodeEncodeError as e:
            print(f"Encoding error with {encoding}: {e}")
        except UnicodeDecodeError as e:
            print(f"Decoding error with {encoding}: {e}")

        print()

encoding_examples()
```

Output:

```
Successfully wrote with utf-8 encoding
Read back: Hello, 世界! Café résumé naïve

Encoding error with ascii: 'ascii' codec can't encode character '世' in position 7: ordin

Successfully wrote with latin-1 encoding
Read back: Hello, ä¸ç! Café résumé naïve
```

8.8 Using the File Pointer ('seek')

The file pointer determines where the next read or write operation will occur in a file.

Example 13: File Pointer Operations

```
def file_pointer_demo():
    # Create a test file
    test_file = 'pointer_demo.txt'
    content = "0123456789\nABCDEFGHIJ\nKLMNOPQRST"

    with open(test_file, 'w') as f:
        f.write(content)

    print(f"Created file with content:\n{content}\n")

    # Demonstrate seek and tell operations
    with open(test_file, 'r') as f:
        # Initial position
        pos = f.tell()
        print(f"Initial position: {pos}")

        # Read 5 characters
        data = f.read(5)
        print(f"Read 5 chars: '{data}'")
        print(f"Current position: {f.tell()}")

        # Seek to position 10
        f.seek(10)
        print(f"After seek(10): {f.tell()}")
        data = f.read(5)
        print(f"Read 5 chars from pos 10: '{data}'")

        # Seek from end
```

```
        f.seek(-5, 2)  # 5 characters from end
        print(f"Position after seek(-5, 2): {f.tell()}")
        data = f.read()
        print(f"Read from -5 to end: '{data}'")

        # Seek from current position
        f.seek(0)  # Go to beginning
        f.read(7)  # Read first 7 characters
        f.seek(3, 1)  # Move 3 positions forward from current
        print(f"Position after seek(3, 1): {f.tell()}")
        data = f.read(3)
        print(f"Read 3 chars: '{data}'")

file_pointer_demo()
```

Output:

```
Created file with content:
0123456789
ABCDEFGHIJ
KLMNOPQRST

Initial position: 0
Read 5 chars: '01234'
Current position: 5
After seek(10): 10
Read 5 chars from pos 10: '\nABCD'
Position after seek(-5, 2): 17
Read from -5 to end: 'PQRST'
Position after seek(3, 1): 10
Read 3 chars: '\nAB'
```

Example 14: Binary File Seeking

```
import struct

def binary_seek_demo():
    # Create a binary file with integers
    numbers = [10, 20, 30, 40, 50]
    binary_file = 'numbers_seek.bin'

    # Write integers to binary file
    with open(binary_file, 'wb') as f:
        for num in numbers:
            f.write(struct.pack('i', num))

    print(f"Created binary file with numbers: {numbers}")

    # Read integers using seek
    with open(binary_file, 'rb') as f:
        # Each integer takes 4 bytes
        int_size = 4

        # Read third integer (index 2)
```

```
        f.seek(2 * int_size)  # Position at 3rd integer
        data = f.read(int_size)
        third_num = struct.unpack('i', data)[^0]
        print(f"Third number (index 2): {third_num}")

        # Read last integer
        f.seek(-int_size, 2)  # Go to last integer
        data = f.read(int_size)
        last_num = struct.unpack('i', data)[^0]
        print(f"Last number: {last_num}")

        # Read all integers using seek
        print("All numbers using seek:")
        for i in range(len(numbers)):
            f.seek(i * int_size)
            data = f.read(int_size)
            num = struct.unpack('i', data)[^0]
            print(f"  Position {i}: {num}")

binary_seek_demo()
```

Output:

```
Created binary file with numbers: [10, 20, 30, 40, 50]
Third number (index 2): 30
Last number: 50
All numbers using seek:
  Position 0: 10
  Position 1: 20
  Position 2: 30
  Position 3: 40
  Position 4: 50
```

8.9 Reading Text into the RPN Project

8.9.1 The RPN Interpreter to Date

Example 15: Basic RPN Calculator

```
class RPNCalculator:
    def __init__(self):
        self.stack = []
        self.variables = {}

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

    def pop(self):
        """Pop and return the top value from stack"""
        if not self.stack:
            raise ValueError("Stack is empty")
```

```
        return self.stack.pop()

    def execute_operation(self, operation):
        """Execute an RPN operation"""
        if operation == '+':
            b, a = self.pop(), self.pop()
            self.push(a + b)
        elif operation == '-':
            b, a = self.pop(), self.pop()
            self.push(a - b)
        elif operation == '*':
            b, a = self.pop(), self.pop()
            self.push(a * b)
        elif operation == '/':
            b, a = self.pop(), self.pop()
            if b == 0:
                raise ValueError("Division by zero")
            self.push(a / b)
        elif operation == 'dup':
            if self.stack:
                self.push(self.stack[-1])
        elif operation == 'swap':
            if len(self.stack) &gt;= 2:
                self.stack[-1], self.stack[-2] = self.stack[-2], self.stack[-1]
        elif operation == 'drop':
            if self.stack:
                self.pop()
        else:
            raise ValueError(f"Unknown operation: {operation}")

    def process_token(self, token):
        """Process a single RPN token"""
        operations = {'+', '-', '*', '/', 'dup', 'swap', 'drop'}

        if token in operations:
            self.execute_operation(token)
        else:
            try:
                # Try to convert to number and push
                self.push(float(token))
            except ValueError:
                raise ValueError(f"Invalid token: {token}")

    def evaluate(self, expression):
        """Evaluate an RPN expression"""
        tokens = expression.split()
        for token in tokens:
            self.process_token(token)

        if len(self.stack) != 1:
            raise ValueError("Invalid RPN expression")

        return self.stack[^0]

    def get_stack(self):
        """Return current stack state"""
```

```
        return self.stack.copy()

# Test the RPN calculator
def test_rpn():
    calc = RPNCalculator()

    # Test expressions
    expressions = [
        "3 4 +",           # 3 + 4 = 7
        "15 7 1 1 + - / 3 * 2 1 1 + + -",  # Complex expression
        "5 1 2 + 4 * + 3 -",  # 5 + ((1 + 2) * 4) - 3 = 14
    ]

    for expr in expressions:
        calc = RPNCalculator()  # Fresh calculator for each test
        try:
            result = calc.evaluate(expr)
            print(f"Expression: {expr}")
            print(f"Result: {result}")
            print()
        except Exception as e:
            print(f"Error evaluating '{expr}': {e}")

test_rpn()
```

Output:

```
Expression: 3 4 +
Result: 7.0

Expression: 15 7 1 1 + - / 3 * 2 1 1 + + -
Result: 5.0

Expression: 5 1 2 + 4 * + 3 -
Result: 14.0
```

8.9.2 Reading RPN from a Text File

Example 16: RPN Calculator with File Input

```
class FileRPNCalculator(RPNCalculator):
    def load_program_from_file(self, filename):
        """Load RPN program from a text file"""
        try:
            with open(filename, 'r') as f:
                program = f.read()
            return program
        except FileNotFoundError:
            raise FileNotFoundError(f"RPN program file '{filename}' not found")

    def execute_file(self, filename):
        """Execute RPN program from file"""
        program = self.load_program_from_file(filename)
```

```
        lines = program.strip().split('\n')

        print(f"Executing RPN program from {filename}:")
        print("-" * 40)

        for line_num, line in enumerate(lines, 1):
            line = line.strip()
            if line and not line.startswith('#'):  # Skip empty lines and comments
                print(f"Line {line_num}: {line}")
                try:
                    tokens = line.split()
                    for token in tokens:
                        self.process_token(token)
                    print(f"Stack after line {line_num}: {self.get_stack()}")
                except Exception as e:
                    print(f"Error on line {line_num}: {e}")
                    break
                print()

        print(f"Final result: {self.stack[-1] if self.stack else 'No result'}")

# Create sample RPN programs
def create_rpn_files():
    # Simple calculation program
    simple_program = """# Simple RPN calculation
# Calculate (3 + 4) * 2
3 4 +
2 *
"""

    with open('simple_rpn.txt', 'w') as f:
        f.write(simple_program)

    # More complex program
    complex_program = """# Complex RPN calculation
# Calculate: (10 + 5) * 3 - 8 / 2
10 5 +
3 *
8 2 /
-
"""

    with open('complex_rpn.txt', 'w') as f:
        f.write(complex_program)

    print("Created RPN program files")

# Test file-based RPN
def test_file_rpn():
    create_rpn_files()

    calc = FileRPNCalculator()

    # Test simple program
    print("Testing simple RPN program:")
    calc.execute_file('simple_rpn.txt')
```

```
    print("\n" + "="*50 + "\n")

    # Test complex program
    calc = FileRPNCalculator()  # Fresh calculator
    print("Testing complex RPN program:")
    calc.execute_file('complex_rpn.txt')

test_file_rpn()
```

Output:

```
Created RPN program files
Testing simple RPN program:
Executing RPN program from simple_rpn.txt:
----------------------------------------
Line 2: 3 4 +
Stack after line 2: [7.0]

Line 3: 2 *
Stack after line 3: [14.0]

Final result: 14.0

==================================================

Testing complex RPN program:
Executing RPN program from complex_rpn.txt:
----------------------------------------
Line 2: 10 5 +
Stack after line 2: [15.0]

Line 3: 3 *
Stack after line 3: [45.0]

Line 4: 8 2 /
Stack after line 4: [45.0, 4.0]

Line 5: -
Stack after line 5: [41.0]

Final result: 41.0
```

8.9.3 Adding an Assignment Operator to RPN

Example 17: Enhanced RPN Calculator with Variables

```
class EnhancedRPNCalculator(FileRPNCalculator):
    def __init__(self):
        super().__init__()
        self.variables = {}

    def execute_operation(self, operation):
```

```
        """Execute an RPN operation including assignment"""
        if operation.startswith('='):
            # Assignment operation: =varname
            var_name = operation[1:]
            if not self.stack:
                raise ValueError("Cannot assign: stack is empty")
            value = self.pop()
            self.variables[var_name] = value
            print(f"Assigned {value} to variable '{var_name}'")

        elif operation.startswith('@'):
            # Variable recall: @varname
            var_name = operation[1:]
            if var_name not in self.variables:
                raise ValueError(f"Unknown variable: {var_name}")
            self.push(self.variables[var_name])
            print(f"Recalled variable '{var_name}': {self.variables[var_name]}")

        elif operation == 'vars':
            # Display all variables
            print("Current variables:")
            for var_name, value in self.variables.items():
                print(f"  {var_name} = {value}")

        else:
            # Use parent class for standard operations
            super().execute_operation(operation)

    def process_token(self, token):
        """Process a single RPN token including variables"""
        operations = {'+', '-', '*', '/', 'dup', 'swap', 'drop', 'vars'}

        if token in operations or token.startswith('=') or token.startswith('@'):
            self.execute_operation(token)
        else:
            try:
                # Try to convert to number and push
                self.push(float(token))
            except ValueError:
                raise ValueError(f"Invalid token: {token}")

# Create enhanced RPN program with variables
def create_enhanced_rpn_program():
    program = """# Enhanced RPN with variables
# Calculate area and perimeter of rectangle
# Width = 5, Height = 3

5 =width      # Store width
3 =height     # Store height

# Calculate area: width * height
@width @height *
=area         # Store area

# Calculate perimeter: 2 * (width + height)
@width @height +
```

```
2 *
=perimeter    # Store perimeter

# Display results
@area
@perimeter
vars          # Show all variables
"""

    with open('enhanced_rpn.txt', 'w') as f:
        f.write(program)

    print("Created enhanced RPN program")

# Test enhanced RPN
def test_enhanced_rpn():
    create_enhanced_rpn_program()

    calc = EnhancedRPNCalculator()
    calc.execute_file('enhanced_rpn.txt')

test_enhanced_rpn()
```

Output:

```
Created enhanced RPN program
Executing RPN program from enhanced_rpn.txt:
----------------------------------------
Line 5: 5 =width
Assigned 5.0 to variable 'width'
Stack after line 5: []

Line 6: 3 =height
Assigned 3.0 to variable 'height'
Stack after line 6: []

Line 9: @width @height *
Recalled variable 'width': 5.0
Recalled variable 'height': 3.0
Stack after line 9: [15.0]

Line 10: =area
Assigned 15.0 to variable 'area'
Stack after line 10: []

Line 13: @width @height +
Recalled variable 'width': 5.0
Recalled variable 'height': 3.0
Stack after line 13: [8.0]

Line 14: 2 *
Stack after line 14: [16.0]

Line 15: =perimeter
Assigned 16.0 to variable 'perimeter'
```

```
Stack after line 15: []

Line 18: @area
Recalled variable 'area': 15.0
Stack after line 18: [15.0]

Line 19: @perimeter
Recalled variable 'perimeter': 16.0
Stack after line 19: [15.0, 16.0]

Line 20: vars
Current variables:
  width = 5.0
  height = 3.0
  area = 15.0
  perimeter = 16.0
Stack after line 20: [15.0, 16.0]

Final result: 16.0
```

8.10 Direct Binary Read/Write

Direct binary operations allow you to work with raw bytes for maximum control over data storage.

Example 18: Direct Binary Operations

```
def direct_binary_operations():
    # Writing different data types as bytes
    filename = 'direct_binary.bin'

    with open(filename, 'wb') as f:
        # Write an integer (4 bytes, little-endian)
        number = 1234
        f.write(number.to_bytes(4, 'little'))

        # Write a float as bytes (IEEE 754 format)
        import struct
        pi = 3.14159
        f.write(struct.pack('f', pi))

        # Write a string as UTF-8 bytes
        text = "Hello"
        f.write(text.encode('utf-8'))

        # Write raw bytes
        raw_data = bytes([65, 66, 67, 68])  # ABCD in ASCII
        f.write(raw_data)

    print(f"Written binary data to {filename}")

    # Reading the binary data back
    with open(filename, 'rb') as f:
        # Read integer
        int_bytes = f.read(4)
        number = int.from_bytes(int_bytes, 'little')
```

```
        print(f"Integer: {number}")

        # Read float
        float_bytes = f.read(4)
        pi = struct.unpack('f', float_bytes)[^0]
        print(f"Float: {pi:.5f}")

        # Read string (5 bytes for "Hello")
        text_bytes = f.read(5)
        text = text_bytes.decode('utf-8')
        print(f"String: {text}")

        # Read raw bytes
        raw_bytes = f.read(4)
        print(f"Raw bytes: {list(raw_bytes)} -&gt; {raw_bytes.decode('ascii')}")

direct_binary_operations()
```

Output:

```
Written binary data to direct_binary.bin
Integer: 1234
Float: 3.14159
String: Hello
Raw bytes: [65, 66, 67, 68] -&gt; ABCD
```

8.11 Converting Data to Fixed-Length Fields ('struct')

The `struct` module provides a powerful way to convert between Python objects and C-style binary data.

8.11.1 Writing and Reading One Number at a Time

Example 19: Single Number Operations with Struct

```
import struct

def single_number_struct():
    filename = 'single_numbers.bin'

    # Different number formats
    numbers = [
        (42, 'i'),           # signed integer (4 bytes)
        (3.14159, 'f'),      # float (4 bytes)
        (2.71828, 'd'),      # double (8 bytes)
        (65535, 'H'),        # unsigned short (2 bytes)
    ]

    # Write numbers
    with open(filename, 'wb') as f:
        for number, format_char in numbers:
            binary_data = struct.pack(format_char, number)
            f.write(binary_data)
```

```
            print(f"Wrote {number} using format '{format_char}' ({len(binary_data)} bytes

    print(f"\nBinary data written to {filename}")

    # Read numbers back
    with open(filename, 'rb') as f:
        print("\nReading numbers back:")

        # Read integer
        data = f.read(struct.calcsize('i'))
        value = struct.unpack('i', data)[^0]
        print(f"Integer: {value}")

        # Read float
        data = f.read(struct.calcsize('f'))
        value = struct.unpack('f', data)[^0]
        print(f"Float: {value:.5f}")

        # Read double
        data = f.read(struct.calcsize('d'))
        value = struct.unpack('d', data)[^0]
        print(f"Double: {value:.5f}")

        # Read unsigned short
        data = f.read(struct.calcsize('H'))
        value = struct.unpack('H', data)[^0]
        print(f"Unsigned short: {value}")

single_number_struct()
```

Output:

```
Wrote 42 using format 'i' (4 bytes)
Wrote 3.14159 using format 'f' (4 bytes)
Wrote 2.71828 using format 'd' (8 bytes)
Wrote 65535 using format 'H' (2 bytes)

Binary data written to single_numbers.bin

Reading numbers back:
Integer: 42
Float: 3.14159
Double: 2.71828
Unsigned short: 65535
```

8.11.2 Writing and Reading Several Numbers at a Time

Example 20: Multiple Numbers with Struct

```
def multiple_numbers_struct():
    filename = 'multiple_numbers.bin'

    # Pack multiple integers at once
```

```
    integers = [10, 20, 30, 40, 50]
    floats = [1.1, 2.2, 3.3, 4.4]

    with open(filename, 'wb') as f:
        # Pack 5 integers using '5i' format
        int_data = struct.pack('5i', *integers)
        f.write(int_data)
        print(f"Packed {len(integers)} integers: {integers}")

        # Pack 4 floats using '4f' format
        float_data = struct.pack('4f', *floats)
        f.write(float_data)
        print(f"Packed {len(floats)} floats: {floats}")

    # Read multiple numbers back
    with open(filename, 'rb') as f:
        # Read 5 integers
        int_size = struct.calcsize('5i')
        int_data = f.read(int_size)
        unpacked_ints = struct.unpack('5i', int_data)
        print(f"\nUnpacked integers: {list(unpacked_ints)}")

        # Read 4 floats
        float_size = struct.calcsize('4f')
        float_data = f.read(float_size)
        unpacked_floats = struct.unpack('4f', float_data)
        print(f"Unpacked floats: {[round(f, 1) for f in unpacked_floats]}")

multiple_numbers_struct()
```

Output:

```
Packed 5 integers: [10, 20, 30, 40, 50]
Packed 4 floats: [1.1, 2.2, 3.3, 4.4]

Unpacked integers: [10, 20, 30, 40, 50]
Unpacked floats: [1.1, 2.2, 3.3, 4.4]
```

8.11.3 Writing and Reading a Fixed-Length String

Example 21: Fixed-Length Strings with Struct

```
def fixed_length_strings():
    filename = 'fixed_strings.bin'

    # Fixed-length string operations
    strings = ["Hello", "World", "Python", "Struct"]
    string_length = 10  # Fixed length of 10 characters

    with open(filename, 'wb') as f:
        for s in strings:
            # Pad or truncate to fixed length
            fixed_string = s.ljust(string_length)[:string_length]
```

```
            # Pack as fixed-length string
            data = struct.pack(f'{string_length}s', fixed_string.encode('utf-8'))
            f.write(data)
            print(f"Packed '{s}' as '{fixed_string}' ({len(data)} bytes)")

    # Read fixed-length strings back
    with open(filename, 'rb') as f:
        print("\nReading fixed-length strings:")
        string_size = struct.calcsize(f'{string_length}s')

        for i in range(len(strings)):
            data = f.read(string_size)
            unpacked = struct.unpack(f'{string_length}s', data)[^0]
            # Decode and strip padding
            decoded_string = unpacked.decode('utf-8').rstrip()
            print(f"String {i+1}: '{decoded_string}'")

fixed_length_strings()
```

Output:

```
Packed 'Hello' as 'Hello     ' (10 bytes)
Packed 'World' as 'World     ' (10 bytes)
Packed 'Python' as 'Python    ' (10 bytes)
Packed 'Struct' as 'Struct    ' (10 bytes)

Reading fixed-length strings:
String 1: 'Hello'
String 2: 'World'
String 3: 'Python'
String 4: 'Struct'
```

8.11.4 Writing and Reading a Variable-Length String

Example 22: Variable-Length Strings with Length Prefix

```
def variable_length_strings():
    filename = 'variable_strings.bin'

    strings = ["Hi", "Hello World!", "Python Programming", "Variable Length"]

    with open(filename, 'wb') as f:
        for s in strings:
            # Encode string to bytes
            string_bytes = s.encode('utf-8')
            length = len(string_bytes)

            # Write length first (as unsigned integer), then string
            f.write(struct.pack('I', length))  # 4-byte length prefix
            f.write(string_bytes)
            print(f"Wrote '{s}' (length: {length})")

    # Read variable-length strings back
```

```
    with open(filename, 'rb') as f:
        print("\nReading variable-length strings:")

        while True:
            # Try to read length prefix
            length_data = f.read(4)
            if len(length_data) &lt; 4:
                break  # End of file

            length = struct.unpack('I', length_data)[^0]
            # Read string data
            string_data = f.read(length)
            decoded_string = string_data.decode('utf-8')
            print(f"Read string (length {length}): '{decoded_string}'")

variable_length_strings()
```

Output:

```
Wrote 'Hi' (length: 2)
Wrote 'Hello World!' (length: 12)
Wrote 'Python Programming' (length: 18)
Wrote 'Variable Length' (length: 15)

Reading variable-length strings:
Read string (length 2): 'Hi'
Read string (length 12): 'Hello World!'
Read string (length 18): 'Python Programming'
Read string (length 15): 'Variable Length'
```

8.11.5 Writing and Reading Strings and Numerics Together

Example 23: Mixed Data Types with Struct

```
def mixed_data_struct():
    filename = 'mixed_data.bin'

    # Record structure: ID (int), Score (float), Name (20-char string), Active (bool)
    records = [
        (1001, 95.5, "Alice Johnson", True),
        (1002, 87.3, "Bob Smith", False),
        (1003, 92.1, "Carol Davis", True),
    ]

    record_format = 'I f 20s ?'  # unsigned int, float, 20-char string, boolean

    with open(filename, 'wb') as f:
        for record_id, score, name, active in records:
            # Prepare the name (pad to 20 characters)
            name_bytes = name.encode('utf-8')[:20].ljust(20, b'\0')

            # Pack the entire record
            packed_data = struct.pack(record_format, record_id, score, name_bytes, active
```

```
            f.write(packed_data)
            print(f"Wrote record: ID={record_id}, Score={score}, Name='{name}', Active={a

    print(f"\nRecord size: {struct.calcsize(record_format)} bytes")

    # Read records back
    with open(filename, 'rb') as f:
        print("\nReading records:")
        record_size = struct.calcsize(record_format)

        record_num = 1
        while True:
            data = f.read(record_size)
            if len(data) &lt; record_size:
                break

            # Unpack the record
            record_id, score, name_bytes, active = struct.unpack(record_format, data)
            # Clean up the name
            name = name_bytes.rstrip(b'\0').decode('utf-8')

            print(f"Record {record_num}: ID={record_id}, Score={score:.1f}, Name='{name}'
            record_num += 1

mixed_data_struct()
```

Output:

```
Wrote record: ID=1001, Score=95.5, Name='Alice Johnson', Active=True
Wrote record: ID=1002, Score=87.3, Name='Bob Smith', Active=False
Wrote record: ID=1003, Score=92.1, Name='Carol Davis', Active=True

Record size: 29 bytes

Reading records:
Record 1: ID=1001, Score=95.5, Name='Alice Johnson', Active=True
Record 2: ID=1002, Score=87.3, Name='Bob Smith', Active=False
Record 3: ID=1003, Score=92.1, Name='Carol Davis', Active=True
```

8.11.6 Low-Level Details: Big Endian Versus Little Endian

Example 24: Endianness Demonstration

```
def endianness_demo():
    number = 0x12345678  # Hexadecimal number for clear byte visualization

    print(f"Number: {number} (0x{number:08x})")
    print()

    # Different endianness formats
    endian_formats = {
        '@': 'Native (system default)',
        '&lt;': 'Little-endian',
```

```
        '&gt;': 'Big-endian',
        '=': 'Native (no alignment)',
        '!': 'Network (big-endian)'
    }

    for prefix, description in endian_formats.items():
        packed = struct.pack(f'{prefix}I', number)
        bytes_list = list(packed)
        hex_bytes = [f'0x{b:02x}' for b in bytes_list]

        print(f"{description} ({prefix}I):")
        print(f"  Bytes: {hex_bytes}")

        # Unpack to verify
        unpacked = struct.unpack(f'{prefix}I', packed)[^0]
        print(f"  Unpacked: {unpacked} (0x{unpacked:08x})")
        print()

    # Demonstrate cross-platform compatibility issue
    print("Cross-platform compatibility test:")

    # Create data with different endianness
    little_endian_data = struct.pack('&lt;I', number)
    big_endian_data = struct.pack('&gt;I', number)

    print(f"Little-endian bytes: {[hex(b) for b in little_endian_data]}")
    print(f"Big-endian bytes: {[hex(b) for b in big_endian_data]}")

    # Try reading with wrong endianness
    wrong_little = struct.unpack('&lt;I', big_endian_data)[^0]
    wrong_big = struct.unpack('&gt;I', little_endian_data)[^0]

    print(f"Reading big-endian data as little-endian: {wrong_little:08x}")
    print(f"Reading little-endian data as big-endian: {wrong_big:08x}")

endianness_demo()
```

Output:

```
Number: 305419896 (0x12345678)

Native (system default) (@I):
  Bytes: ['0x78', '0x56', '0x34', '0x12']
  Unpacked: 305419896 (0x12345678)

Little-endian (&lt;I):
  Bytes: ['0x78', '0x56', '0x34', '0x12']
  Unpacked: 305419896 (0x12345678)

Big-endian (&gt;I):
  Bytes: ['0x12', '0x34', '0x56', '0x78']
  Unpacked: 305419896 (0x12345678)

Native (no alignment) (=I):
  Bytes: ['0x78', '0x56', '0x34', '0x12']
```

```
  Unpacked: 305419896 (0x12345678)

Network (big-endian) (!I):
  Bytes: ['0x12', '0x34', '0x56', '0x78']
  Unpacked: 305419896 (0x12345678)

Cross-platform compatibility test:
Little-endian bytes: ['0x78', '0x56', '0x34', '0x12']
Big-endian bytes: ['0x12', '0x34', '0x56', '0x78']
Reading big-endian data as little-endian: 78563412
Reading little-endian data as big-endian: 78563412
```

8.12 Using the Pickling Package

Python's `pickle` module provides a way to serialize Python objects into binary format.

Example 25: Basic Pickling Operations

```
import pickle

def basic_pickling():
    # Various Python objects to pickle
    data_to_pickle = {
        'integer': 42,
        'float': 3.14159,
        'string': 'Hello, Pickle!',
        'list': [1, 2, 3, 'four', 5.0],
        'dict': {'name': 'Alice', 'age': 30, 'scores': [95, 87, 92]},
        'tuple': (1, 'two', 3.0),
        'set': {1, 2, 3, 4, 5}
    }

    # Pickle data to file
    with open('pickled_data.pkl', 'wb') as f:
        pickle.dump(data_to_pickle, f)

    print("Data pickled successfully!")
    print(f"Original data: {data_to_pickle}")

    # Unpickle data from file
    with open('pickled_data.pkl', 'rb') as f:
        unpickled_data = pickle.load(f)

    print(f"\nUnpickled data: {unpickled_data}")
    print(f"Data types preserved: {type(unpickled_data['list'])}, {type(unpickled_data['s

    # Verify data integrity
    print(f"Data integrity check: {data_to_pickle == unpickled_data}")

basic_pickling()
```

Output:

```
Data pickled successfully!
Original data: {'integer': 42, 'float': 3.14159, 'string': 'Hello, Pickle!', 'list': [1,

Unpickled data: {'integer': 42, 'float': 3.14159, 'string': 'Hello, Pickle!', 'list': [1,
Data types preserved: &lt;class 'list'&gt;, &lt;class 'set'&gt;
Data integrity check: True
```

Example 26: Pickling Custom Objects

```
class Student:
    def __init__(self, name, age, grades):
        self.name = name
        self.age = age
        self.grades = grades

    def average_grade(self):
        return sum(self.grades) / len(self.grades) if self.grades else 0

    def __str__(self):
        return f"Student(name='{self.name}', age={self.age}, grades={self.grades})"

def pickle_custom_objects():
    # Create custom objects
    students = [
        Student("Alice", 20, [85, 92, 78, 96]),
        Student("Bob", 19, [88, 84, 91, 87]),
        Student("Carol", 21, [95, 89, 93, 90])
    ]

    print("Original students:")
    for student in students:
        print(f"  {student} - Average: {student.average_grade():.1f}")

    # Pickle the list of objects
    with open('students.pkl', 'wb') as f:
        pickle.dump(students, f)

    print("\nStudents pickled successfully!")

    # Unpickle the objects
    with open('students.pkl', 'rb') as f:
        unpickled_students = pickle.load(f)

    print("\nUnpickled students:")
    for student in unpickled_students:
        print(f"  {student} - Average: {student.average_grade():.1f}")

    # Verify methods still work
    print(f"\nFirst student's average grade: {unpickled_students[^0].average_grade():.1f}

pickle_custom_objects()
```

Output:

```
Original students:
  Student(name='Alice', age=20, grades=[85, 92, 78, 96]) - Average: 87.8
  Student(name='Bob', age=19, grades=[88, 84, 91, 87]) - Average: 87.5
  Student(name='Carol', age=21, grades=[95, 89, 93, 90]) - Average: 91.8

Students pickled successfully!

Unpickled students:
  Student(name='Alice', age=20, grades=[85, 92, 78, 96]) - Average: 87.8
  Student(name='Bob', age=19, grades=[88, 84, 91, 87]) - Average: 87.5
  Student(name='Carol', age=21, grades=[95, 89, 93, 90]) - Average: 91.8

First student's average grade: 87.8
```

8.13 Using the 'shelve' Package

The `shelve` module provides a persistent dictionary-like object using pickle for storage.

Example 27: Basic Shelve Operations

```
import shelve

def basic_shelve_operations():
    # Open a shelf (creates database files)
    with shelve.open('data_shelf') as shelf:
        # Store various types of data
        shelf['numbers'] = [1, 2, 3, 4, 5]
        shelf['message'] = 'Hello, Shelve!'
        shelf['user_data'] = {
            'name': 'Alice',
            'email': 'alice@example.com',
            'preferences': ['python', 'data_science', 'machine_learning']
        }
        shelf['pi'] = 3.14159

        print("Data stored in shelf:")
        for key in shelf:
            print(f"  {key}: {shelf[key]}")

    # Reopen shelf to demonstrate persistence
    print("\nReopening shelf to verify persistence:")
    with shelve.open('data_shelf') as shelf:
        print(f"Available keys: {list(shelf.keys())}")
        print(f"Message: {shelf['message']}")
        print(f"User data: {shelf['user_data']}")

        # Modify existing data
        numbers = shelf['numbers']
        numbers.append(6)
        shelf['numbers'] = numbers  # Must reassign for shelve to save changes

        print(f"Updated numbers: {shelf['numbers']}")
```

```
basic_shelve_operations()
```

Output:

```
Data stored in shelf:
  numbers: [1, 2, 3, 4, 5]
  message: Hello, Shelve!
  user_data: {'name': 'Alice', 'email': 'alice@example.com', 'preferences': ['python', 'd
  pi: 3.14159

Reopening shelf to verify persistence:
Available keys: ['numbers', 'message', 'user_data', 'pi']
Message: Hello, Shelve!
User data: {'name': 'Alice', 'email': 'alice@example.com', 'preferences': ['python', 'dat
Updated numbers: [1, 2, 3, 4, 5, 6]
```

Example 28: Advanced Shelve Usage with Custom Objects

```
class Person:
    def __init__(self, name, age, occupation):
        self.name = name
        self.age = age
        self.occupation = occupation
        self.friends = []

    def add_friend(self, friend_name):
        if friend_name not in self.friends:
            self.friends.append(friend_name)

    def __str__(self):
        return f"Person(name='{self.name}', age={self.age}, occupation='{self.occupation}

def advanced_shelve_usage():
    # Create a people database using shelve
    with shelve.open('people_db') as db:
        # Add people to the database
        alice = Person("Alice Johnson", 28, "Data Scientist")
        alice.add_friend("Bob")
        alice.add_friend("Carol")

        bob = Person("Bob Smith", 32, "Software Engineer")
        bob.add_friend("Alice")

        carol = Person("Carol Davis", 25, "Designer")
        carol.add_friend("Alice")

        # Store in database using names as keys
        db['alice'] = alice
        db['bob'] = bob
        db['carol'] = carol

        print("People added to database:")
        for key, person in db.items():
```

```
            print(f"  {key}: {person}")
            print(f"    Friends: {person.friends}")

    # Query the database
    print("\nQuerying the people database:")
    with shelve.open('people_db') as db:
        # Find people by criteria
        print("People over 30:")
        for key, person in db.items():
            if person.age &gt; 30:
                print(f"  {person.name} (age {person.age})")

        # Update a person's data
        if 'alice' in db:
            alice = db['alice']
            alice.add_friend("David")
            db['alice'] = alice  # Save changes
            print(f"\nUpdated Alice's friends: {alice.friends}")

        # Database statistics
        print(f"\nDatabase statistics:")
        print(f"  Total people: {len(db)}")
        print(f"  Average age: {sum(person.age for person in db.values()) / len(db):.1f}"

advanced_shelve_usage()
```

Output:

```
People added to database:
  alice: Person(name='Alice Johnson', age=28, occupation='Data Scientist')
    Friends: ['Bob', 'Carol']
  bob: Person(name='Bob Smith', age=32, occupation='Software Engineer')
    Friends: ['Alice']
  carol: Person(name='Carol Davis', age=25, occupation='Designer')
    Friends: ['Alice']

Querying the people database:
People over 30:
  Bob Smith (age 32)

Updated Alice's friends: ['Bob', 'Carol', 'David']

Database statistics:
  Total people: 3
  Average age: 28.3
```

Chapter Summary

Chapter 8 covered comprehensive file handling techniques in Python:

1. **Text vs Binary Files**: Understanding the fundamental differences and when to use each type
2. **File Operations**: Basic reading, writing, and file management operations

3. **Exception Handling**: Proper error handling for robust file operations
4. **Context Managers**: Using the `with` statement for automatic resource management
5. **Advanced Text Processing**: Complex text file operations and encoding handling
6. **File Positioning**: Using seek and tell for precise file navigation
7. **RPN Calculator Project**: Practical application demonstrating file-based program execution
8. **Binary Data Handling**: Direct byte operations and the struct module
9. **Serialization**: Using pickle for object persistence
10. **Database-like Storage**: Using shelve for persistent dictionary-like data storage