Lecture 10 Handout

File Processing and Data Formats

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 9, Sections 9.1-9.12 (Files and Exceptions)
Reference Notebooks: `ch09/09_01.ipynb` (Introduction), `ch09/09_03.*.ipynb` (Text Files), `ch09/09_05.ipynb` (JSON), `ch09/09_12.*.ipynb` (CSV Files)

Learning Objectives

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

1. **Understand file concepts** and persistent data storage principles
2. **Master text file operations** reading, writing, and updating text files
3. **Work with CSV files** using both manual parsing and csv module
4. **Handle JSON data** serialization and deserialization for data interchange
5. **Use context managers** with statement for proper resource management
6. **Process structured data** from files into appropriate data structures
7. **Implement file exception handling** for robust file operations
8. **Work with pandas basics** for CSV file processing (preview)
9. **Design file-based applications** following professional patterns

Prerequisites Review

Building on Your Comprehensive Programming Foundation:

From Lecture 1: Python basics, variables, all data types, arithmetic operations, print(), input(), f-strings, type conversion
From Lecture 2: Complete if/elif/else structures, boolean logic, string methods for validation (.isdigit(), .strip(), etc.)
From Lecture 3: while/for loops, range() mastery, basic list foundation (creation, indexing, slicing, len()), nested control structures

From Lecture 4: List comprehension mastery, data processing, transformation, mapping, filtering operations
From Lecture 5: Function-oriented programming, def keyword, parameters, return values, modules (random, math), scope
From Lecture 6: Exception handling expertise, try/except/else/finally, defensive programming, custom exceptions
From Lecture 7: Advanced data structures mastery (lists, tuples, dictionaries, sets), selection strategies, nested structures
From Lecture 8: Object-oriented programming basics, class creation, __init__, methods, properties, encapsulation, string representations
From Lecture 9: Advanced OOP with inheritance, polymorphism, operator overloading, duck typing, exception hierarchies

Transformation Goal: Evolve from **in-memory data processing** to **persistent data storage and retrieval** - learning to save data between program runs and work with common data formats.

1 Part 1: Understanding Files and Persistence

1.1 Why Files Matter

Up until now, all the data in our programs has existed only in memory - when the program ends, the data disappears. Files provide persistent storage that survives program termination. Think of the difference between writing notes on a whiteboard (memory) versus writing in a notebook (file) - the whiteboard gets erased, but the notebook keeps your notes permanently.

Files enable us to:

- Save program data between runs (game progress, user settings)
- Share data between different programs (export/import)
- Process large datasets that don't fit in memory
- Create logs and reports for later analysis
- Store configuration and settings

1.2 File Concepts and Terminology

File Path: The location of a file in the file system

- **Absolute path:** Complete path from root (`/Users/alice/documents/data.txt`)
- **Relative path:** Path relative to current directory (`data.txt` or `../data/file.csv`)

File Modes: How we intend to use the file

- **'r':** Read mode (file must exist)
- **'w':** Write mode (creates new file or overwrites existing)
- **'a':** Append mode (adds to end of existing file)
- **'r+':** Read and write mode

2 Part 2: Text File Operations

2.1 The Context Manager Pattern

Python provides the `with` statement to ensure files are properly closed, even if errors occur. This prevents resource leaks and is considered best practice for file operations.

```
# Best practice: Using with statement (context manager)
with open('example.txt', 'r') as file:
    content = file.read()
    # File automatically closes when we exit the with block
# No need to call file.close()

# What NOT to do (but you should understand why)
file = open('example.txt', 'r')
content = file.read()
file.close()  # Easy to forget, won't happen if error occurs
```

2.2 Reading Text Files

There are several ways to read from a text file, each suited to different needs:

```
# Method 1: Read entire file as a single string
with open('story.txt', 'r') as file:
    entire_content = file.read()
    print(f"File contains {len(entire_content)} characters")

# Method 2: Read file line by line into a list
with open('story.txt', 'r') as file:
    lines = file.readlines()
    print(f"File has {len(lines)} lines")

# Method 3: Process file line by line (memory efficient)
with open('story.txt', 'r') as file:
    for line_num, line in enumerate(file, 1):
        print(f"Line {line_num}: {line.strip()}")
```

2.3 Writing to Text Files

Writing creates a new file or overwrites an existing one:

```
# Writing strings to a file
with open('output.txt', 'w') as file:
    file.write("Hello, File World!\n")
    file.write("This is line 2\n")

    # Writing multiple lines at once
    lines = ["Line 3\n", "Line 4\n", "Line 5\n"]
    file.writelines(lines)

# Appending to an existing file
with open('log.txt', 'a') as file:
    from datetime import datetime
    file.write(f"{datetime.now()}: Program started\n")
```

3 Part 3: Working with CSV Files

3.1 CSV Format Overview

CSV (Comma-Separated Values) files store tabular data in plain text. Each line represents a row, and commas separate the columns. It's like a simplified spreadsheet that any program can read.

```
# Example CSV content:
# name,age,grade
# Alice,20,A
# Bob,19,B
# Carol,21,A
```

3.2 Manual CSV Processing

You can process CSV files using basic string operations:

```
# Reading CSV manually
students = []
with open('students.csv', 'r') as file:
    headers = file.readline().strip().split(',')

    for line in file:
        values = line.strip().split(',')
        student = dict(zip(headers, values))
        students.append(student)

# Writing CSV manually
with open('grades.csv', 'w') as file:
    file.write('name,midterm,final,average\n')

    for student in student_grades:
        name = student['name']
        mid = student['midterm']
        final = student['final']
        avg = (mid + final) / 2
        file.write(f'{name},{mid},{final},{avg}\n')
```

3.3 Using the csv Module

The csv module handles edge cases like commas in data, quotes, and different delimiters:

```
import csv

# Reading with csv.reader()
with open('students.csv', 'r') as file:
    csv_reader = csv.reader(file)
    headers = next(csv_reader)  # Skip header row

    for row in csv_reader:
        name, age, grade = row
        print(f"{name} ({age}) earned grade: {grade}")

# Writing with csv.writer()
with open('report.csv', 'w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerow(['Product', 'Price', 'Quantity'])
```

```
    csv_writer.writerow(['Apple', 0.50, 100])
    csv_writer.writerow(['Banana', 0.30, 150])

# Using DictReader for more readable code
with open('students.csv', 'r') as file:
    csv_reader = csv.DictReader(file)

    for row in csv_reader:
        # Access by column name instead of index
        print(f"{row['name']} is {row['age']} years old")
```

4 Part 4: JSON Data Format

4.1 Understanding JSON

JSON (JavaScript Object Notation) is a human-readable format for storing and exchanging structured data. It maps naturally to Python dictionaries and lists, making it perfect for configuration files and data interchange.

```
# JSON example:
{
    "name": "Alice Johnson",
    "age": 20,
    "courses": ["Python", "Data Structures", "Calculus"],
    "gpa": 3.8,
    "active": true
}
```

4.2 JSON Serialization and Deserialization

```
import json

# Python object to JSON (serialization)
student = {
    'name': 'Alice Johnson',
    'age': 20,
    'courses': ['Python', 'Data Structures', 'Calculus'],
    'gpa': 3.8,
    'active': True
}

# Write to JSON file
with open('student.json', 'w') as file:
    json.dump(student, file, indent=4)

# Read from JSON file (deserialization)
with open('student.json', 'r') as file:
    loaded_student = json.load(file)
    print(f"Loaded: {loaded_student['name']}")

# Working with JSON strings
json_string = json.dumps(student)  # Object to string
parsed = json.loads(json_string)   # String to object
```

4.3 JSON Use Cases

- **Configuration files:** Application settings
- **Data exchange:** APIs and web services
- **Data persistence:** Saving complex objects
- **Logging:** Structured log entries

5 Part 5: Exception Handling for Files

5.1 Common File Exceptions

File operations can fail for various reasons. Robust programs handle these gracefully:

```
# Comprehensive file error handling
try:
    with open('data.txt', 'r') as file:
        content = file.read()
        process_data(content)
except FileNotFoundError:
    print("Error: File not found. Creating default file...")
    create_default_file()
except PermissionError:
    print("Error: No permission to read file")
except IOError as e:
    print(f"I/O error occurred: {e}")
except Exception as e:
    print(f"Unexpected error: {e}")
```

5.2 Defensive File Programming

```
import os

def safe_read_file(filename, default_content=''):
    """Read file with comprehensive error handling."""
    # Check if file exists
    if not os.path.exists(filename):
        print(f"Warning: {filename} not found")
        return default_content

    try:
        with open(filename, 'r') as file:
            return file.read()
    except Exception as e:
        print(f"Error reading {filename}: {e}")
        return default_content

def safe_write_file(filename, content):
    """Write file with backup and error handling."""
    # Create backup if file exists
    if os.path.exists(filename):
        backup = filename + '.backup'
        try:
            with open(filename, 'r') as source:
                with open(backup, 'w') as dest:
```

```
                    dest.write(source.read())
        except Exception as e:
            print(f"Warning: Could not create backup: {e}")

    # Write new content
    try:
        with open(filename, 'w') as file:
            file.write(content)
        return True
    except Exception as e:
        print(f"Error writing {filename}: {e}")
        return False
```

6 Part 6: Practical File Applications

6.1 Example 1: Personal Diary Application

```
from datetime import datetime

def write_diary_entry(filename='diary.txt'):
    """Write a diary entry with timestamp."""
    timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

    entry = input("What's on your mind today? ")

    # Append to diary file with timestamp
    with open(filename, 'a') as file:
        file.write(f"\n{'='*50}\n")
        file.write(f"Date: {timestamp}\n")
        file.write(f"Entry: {entry}\n")
        file.write(f"{'='*50}\n")

    print("Diary entry saved!")

def read_diary(filename='diary.txt'):
    """Read all diary entries."""
    try:
        with open(filename, 'r') as file:
            content = file.read()
            print("\nYour Diary Entries:")
            print(content)
    except FileNotFoundError:
        print("No diary entries found yet!")

# Usage
write_diary_entry()
read_diary()
```

6.2 Example 2: CSV Grade Tracker with Letter Grades

```
import csv

def calculate_letter_grade(score):
    """Convert numeric score to letter grade."""
    if score >= 90:
```

```
        return 'A'
    elif score >= 80:
        return 'B'
    elif score >= 70:
        return 'C'
    elif score >= 60:
        return 'D'
    else:
        return 'F'

class GradeTracker:
    """Track student grades in CSV format."""

    def __init__(self, filename='grades.csv'):
        self.filename = filename
        self.initialize_file()

    def initialize_file(self):
        """Create CSV with headers if it doesn't exist."""
        try:
            with open(self.filename, 'x', newline='') as file:
                writer = csv.writer(file)
                writer.writerow(['Name', 'Assignment', 'Score', 'Letter
                    Grade'])
        except FileExistsError:
            pass  # File already exists

    def add_grade(self, name, assignment, score):
        """Add a grade to the CSV file."""
        letter_grade = calculate_letter_grade(score)

        with open(self.filename, 'a', newline='') as file:
            writer = csv.writer(file)
            writer.writerow([name, assignment, score, letter_grade])

        print(f"Grade recorded: {name} - {assignment}: {score} ({
            letter_grade})")

    def view_grades(self):
        """Display all grades from CSV."""
        with open(self.filename, 'r') as file:
            reader = csv.DictReader(file)

            print("\nAll Grades:")
            print("-" * 50)
            for row in reader:
                print(f"{row['Name']:15} {row['Assignment']:15} "
                      f"{row['Score']:>5} ({row['Letter Grade']})")

# Usage
tracker = GradeTracker()
tracker.add_grade('Alice', 'Midterm', 95)
tracker.add_grade('Bob', 'Midterm', 82)
tracker.view_grades()
```

6.3 Example 3: Contact Manager with JSON

```python
import json

class ContactManager:
    """Manage contacts with JSON persistence."""

    def __init__(self, filename='contacts.json'):
        self.filename = filename
        self.contacts = self.load_contacts()

    def load_contacts(self):
        """Load contacts from JSON file."""
        try:
            with open(self.filename, 'r') as file:
                return json.load(file)
        except FileNotFoundError:
            return []

    def save_contacts(self):
        """Save contacts to JSON file."""
        with open(self.filename, 'w') as file:
            json.dump(self.contacts, file, indent=4)

    def add_contact(self, name, phone, email):
        """Add a new contact."""
        contact = {
            'name': name,
            'phone': phone,
            'email': email,
            'id': len(self.contacts) + 1
        }
        self.contacts.append(contact)
        self.save_contacts()
        print(f"Contact added: {name}")

    def search_contacts(self, search_term):
        """Search contacts by name."""
        results = []
        search_lower = search_term.lower()

        for contact in self.contacts:
            if search_lower in contact['name'].lower():
                results.append(contact)

        return results

    def display_contacts(self):
        """Display all contacts."""
        if not self.contacts:
            print("No contacts found.")
            return

        print("\nYour Contacts:")
        print("-" * 60)
        for contact in self.contacts:
            print(f"ID: {contact['id']} | Name: {contact['name']:20} |
                "
```

```
                    f"Phone: {contact['phone']:15} | Email: {contact['
                        email']}")

# Usage
cm = ContactManager()
cm.add_contact('Alice Smith', '555-1234', 'alice@email.com')
cm.add_contact('Bob Johnson', '555-5678', 'bob@email.com')
cm.display_contacts()
```

6.4 Example 4: Comprehensive Student Grade System

```
import json
import csv
from datetime import datetime

class StudentGradeSystem:
    """Complete grade management system with multiple file formats."""

    def __init__(self):
        self.students_file = 'students.json'
        self.grades_file = 'grades.csv'
        self.log_file = 'system.log'
        self.load_data()

    def load_data(self):
        """Load student data from JSON."""
        try:
            with open(self.students_file, 'r') as file:
                self.students = json.load(file)
        except FileNotFoundError:
            self.students = {}
            self.save_students()

    def save_students(self):
        """Save student data to JSON."""
        with open(self.students_file, 'w') as file:
            json.dump(self.students, file, indent=4)

    def log_action(self, action):
        """Log system actions to text file."""
        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        with open(self.log_file, 'a') as file:
            file.write(f"[{timestamp}] {action}\n")

    def add_student(self, student_id, name, email):
        """Add a new student."""
        self.students[student_id] = {
            'name': name,
            'email': email,
            'grades': []
        }
        self.save_students()
        self.log_action(f"Added student: {name} (ID: {student_id})")

    def record_grade(self, student_id, assignment, score):
        """Record a grade and save to CSV."""
        if student_id in self.students:
```

```
            # Update JSON data
            self.students[student_id]['grades'].append({
                'assignment': assignment,
                'score': score,
                'date': datetime.now().isoformat()
            })
            self.save_students()

            # Append to CSV
            with open(self.grades_file, 'a', newline='') as file:
                writer = csv.writer(file)
                writer.writerow([
                    student_id,
                    self.students[student_id]['name'],
                    assignment,
                    score,
                    datetime.now().strftime('%Y-%m-%d')
                ])

            self.log_action(f"Recorded grade: {student_id} - {assignment}: {score}")

    def generate_report(self, student_id):
        """Generate a student report."""
        if student_id not in self.students:
            return "Student not found"

        student = self.students[student_id]
        report = f"\nStudent Report: {student['name']} (ID: {student_id})\n"
        report += f"Email: {student['email']}\n\n"

        if student['grades']:
            report += "Grades:\n"
            total = 0
            for grade in student['grades']:
                report += f"  - {grade['assignment']}: {grade['score']}\n"
                total += grade['score']

            average = total / len(student['grades'])
            report += f"\nAverage Score: {average:.2f}\n"
        else:
            report += "No grades recorded yet.\n"

        return report

# Usage
system = StudentGradeSystem()
system.add_student('S001', 'Alice Johnson', 'alice@university.edu')
system.record_grade('S001', 'Midterm', 85)
system.record_grade('S001', 'Final', 92)
print(system.generate_report('S001'))
```

7 Part 7: Brief Introduction to pandas for CSV

While we'll explore pandas more deeply later, here's a preview of its power for CSV files:

```python
import pandas as pd

# Read CSV into DataFrame
df = pd.read_csv('students.csv')

# Basic operations
print(df.head())                    # Show first 5 rows
print(df.describe())                # Statistical summary
print(df[df['grade'] == 'A'])       # Filter rows

# Save processed data
df.to_csv('processed_students.csv', index=False)
```

Summary and Best Practices

Key Takeaways:

1. **Always use context managers** (`with` statements) for file operations
2. **Handle exceptions** to make your file operations robust
3. **Choose the right format:**
   - Text files for simple unstructured data
   - CSV for tabular data
   - JSON for structured, nested data
4. **Think about encoding** - use UTF-8 for international text
5. **Consider file paths** - use os.path for cross-platform code
6. **Validate data** before writing and after reading
7. **Create backups** when updating important files

Common Pitfalls to Avoid

- Forgetting to close files (use `with` statement)
- Not handling file exceptions
- Using wrong mode ('w' overwrites!)
- Hardcoding file paths
- Not validating file data
- Ignoring encoding issues

Practice Exercises

Exercise 1: Diary Application
Create a personal diary application that:

- Allows users to write diary entries
- Automatically adds timestamps to each entry
- Saves entries to a text file with proper formatting
- Includes functions to read previous entries
- Handles file errors gracefully

Exercise 2: Grade Tracker
Build a CSV-based grade tracking system that:

- Records student names, assignments, and scores
- Automatically calculates letter grades (A, B, C, D, F)
- Saves all data in CSV format with proper headers
- Provides functions to view all grades
- Includes grade statistics (average, highest, lowest)

Exercise 3: Contact Manager
Develop a contact management system using JSON that:

- Stores contact information (name, phone, email)
- Provides add, search, update, and delete functionality
- Persists data using JSON format
- Includes validation for phone numbers and emails
- Displays contacts in a user-friendly format

Exercise 4: Expense Tracker Challenge
Create a comprehensive expense tracking application that:

- Records expenses with date, category, amount, and description
- Uses CSV for transaction storage
- Generates monthly summaries saved as JSON
- Exports readable reports to text files
- Implements data validation and error handling
- Calculates totals by category and time period

Bonus Challenge: Integrate all three file formats (text logs, CSV data, JSON summaries) in your expense tracker for a complete financial management system!

Next Week Preview

In Lecture 11, we'll explore NumPy arrays - a powerful tool for numerical computing that will revolutionize how you work with numerical data!