Lecture 11 Handout

Advanced Data Processing

JSON, Excel, and NumPy Introduction

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2025

Required Reading

Textbook: Chapter 9 (JSON sections), pandas documentation for Excel, Chapter 7 (NumPy introduction)
Reference Notebooks: `ch09/09_05.ipynb` (JSON), pandas Excel tutorials, `ch07/07_01.ipynb` through `ch07/07_03.ipynb` (NumPy basics)

Learning Objectives

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

1. **Master advanced JSON** complex structures, APIs, validation, error handling
2. **Work with Excel files** reading, writing, multiple sheets, formatting with pandas
3. **Understand NumPy fundamentals** arrays vs lists, performance benefits, basic operations
4. **Convert between formats** JSON to Excel, CSV to Excel, array to DataFrame
5. **Build data pipelines** integrating multiple formats for analysis
6. **Handle complex data** nested structures, missing values, type conversions
7. **Apply best practices** for each format's strengths and limitations

Prerequisites Review

Building on Your Complete Programming Foundation:
Building on all concepts from Lectures 1-9, plus from **Lecture 10**:

- File I/O operations with text and CSV formats
- Context managers and exception handling for files
- Basic JSON introduction (json.dumps(), json.loads())

- pandas preview (read_csv(), to_csv())
- Data persistence patterns

Transformation Goal: Evolve from **basic file formats** to **advanced data interchange and analysis** - mastering JSON complexity, Excel business formats, and high-performance NumPy arrays.

1 Part 1: Advanced JSON Processing

1.1 Beyond Basic JSON - Complex Structures

In Lecture 10, you learned basic JSON serialization. Now we'll explore complex, nested JSON structures that mirror real-world data from APIs and configuration systems. JSON's power lies in representing hierarchical data with arbitrary nesting depth.

```
# Complex JSON structure example
import json

# Real-world JSON: Social media post
social_post = {
    "id": "post_12345",
    "author": {
        "username": "alice_dev",
        "name": "Alice Johnson",
        "verified": True,
        "followers": 1250
    },
    "content": {
        "text": "Just learned advanced JSON processing!",
        "media": [
            {"type": "image", "url": "img1.jpg", "alt": "Code
                screenshot"},
            {"type": "image", "url": "img2.jpg", "alt": "Results"}
        ],
        "tags": ["python", "json", "programming"]
    },
    "engagement": {
        "likes": 42,
        "comments": [
            {
                "user": "bob_coder",
                "text": "Great tutorial!",
                "timestamp": "2025-03-15T10:30:00Z"
            },
            {
                "user": "carol_dev",
                "text": "Very helpful, thanks!",
                "timestamp": "2025-03-15T11:45:00Z"
            }
        ]
    },
    "metadata": {
        "created_at": "2025-03-15T09:00:00Z",
        "last_updated": "2025-03-15T11:45:00Z",
        "visibility": "public"
    }
```

```
}

# Pretty printing for readability
print(json.dumps(social_post, indent=2))
```

1.2 JSON Parsing and Navigation

Working with nested JSON requires careful navigation through the structure:

```
# Accessing nested data safely
def safe_get_nested(data, keys, default=None):
    """Safely navigate nested dictionary with list of keys."""
    for key in keys:
        if isinstance(data, dict):
            data = data.get(key, default)
        else:
            return default
    return data

# Examples of navigation
author_name = safe_get_nested(social_post, ['author', 'name'])
first_comment = safe_get_nested(social_post, ['engagement', 'comments',
    0])

# Processing arrays within JSON
media_urls = [item['url'] for item in social_post['content']['media']]
comment_texts = [c['text'] for c in social_post['engagement']['comments
    ']]
```

1.3 JSON Validation and Error Handling

Real-world JSON often contains errors or unexpected structures:

```
# JSON validation function
def validate_json_schema(data, required_fields):
    """Validate JSON has required structure."""
    errors = []

    for field_path in required_fields:
        keys = field_path.split('.')
        value = safe_get_nested(data, keys)
        if value is None:
            errors.append(f"Missing required field: {field_path}")

    return errors

# Robust JSON loading with error handling
def load_json_safely(filename):
    """Load JSON with comprehensive error handling."""
    try:
        with open(filename, 'r') as file:
            data = json.load(file)
        return data, None
    except FileNotFoundError:
        return None, f"File not found: {filename}"
    except json.JSONDecodeError as e:
        return None, f"Invalid JSON: {e}"
```

```
    except Exception as e:
        return None, f"Unexpected error: {e}"
```

1.4 Advanced JSON Features

```
# Custom JSON encoding for non-serializable types
from datetime import datetime

class DateTimeEncoder(json.JSONEncoder):
    """Custom encoder for datetime objects."""
    def default(self, obj):
        if isinstance(obj, datetime):
            return obj.isoformat()
        return super().default(obj)

# Using custom encoder
data_with_dates = {
    "event": "Login",
    "timestamp": datetime.now(),
    "user": "alice"
}

json_str = json.dumps(data_with_dates, cls=DateTimeEncoder, indent=2)

# JSON with configuration options
config = {
    "app_settings": {
        "debug_mode": False,
        "max_connections": 100,
        "timeouts": {
            "connection": 30,
            "read": 60,
            "write": 60
        },
        "features": {
            "auth": {"enabled": True, "providers": ["local", "oauth"]},
            "caching": {"enabled": True, "ttl": 3600}
        }
    }
}
```

2 Part 2: Excel File Handling with pandas

2.1 Introduction to Excel Files

Excel files (.xlsx) are the business world's standard for data exchange. Unlike CSV files, Excel supports multiple worksheets, formatting, formulas, and data types within a single file. pandas makes Excel manipulation straightforward.

```
import pandas as pd
import numpy as np
from datetime import datetime

# Reading Excel files
# Basic read - first sheet by default
df = pd.read_excel('sales_data.xlsx')
```

```
8
9  # Read specific sheet by name or index
10 df_customers = pd.read_excel('sales_data.xlsx', sheet_name='Customers')
11 df_products = pd.read_excel('sales_data.xlsx', sheet_name=1)  # Second
       sheet
12
13 # Read all sheets into a dictionary
14 all_sheets = pd.read_excel('sales_data.xlsx', sheet_name=None)
15 for sheet_name, sheet_df in all_sheets.items():
16     print(f"Sheet: {sheet_name}, Rows: {len(sheet_df)}")
```

2.2 Writing Excel Files with Multiple Sheets

```
1  # Create sample data for multiple sheets
2  students_data = {
3      'StudentID': ['S001', 'S002', 'S003', 'S004', 'S005'],
4      'Name': ['Alice', 'Bob', 'Carol', 'David', 'Eve'],
5      'Major': ['CS', 'Math', 'CS', 'Physics', 'CS'],
6      'Year': [3, 2, 4, 2, 3]
7  }
8
9  grades_data = {
10     'StudentID': ['S001', 'S001', 'S002', 'S002', 'S003'],
11     'Course': ['Python', 'DataSci', 'Python', 'Stats', 'Python'],
12     'Grade': [95, 88, 87, 92, 91],
13     'Semester': ['Fall2024', 'Fall2024', 'Fall2024', 'Fall2024', '
           Fall2024']
14 }
15
16 # Create DataFrames
17 df_students = pd.DataFrame(students_data)
18 df_grades = pd.DataFrame(grades_data)
19
20 # Write to Excel with multiple sheets
21 with pd.ExcelWriter('student_records.xlsx', engine='openpyxl') as
       writer:
22     df_students.to_excel(writer, sheet_name='Students', index=False)
23     df_grades.to_excel(writer, sheet_name='Grades', index=False)
24
25     # Add a summary sheet
26     summary = df_grades.groupby('StudentID')['Grade'].agg(['mean', '
           count'])
27     summary.to_excel(writer, sheet_name='Summary')
```

2.3 Excel Formatting and Styling

```
1  # Advanced Excel formatting
2  def create_formatted_report(data, filename):
3      """Create Excel report with formatting."""
4      with pd.ExcelWriter(filename, engine='openpyxl') as writer:
5          # Write data
6          data.to_excel(writer, sheet_name='Report', index=False)
7
8          # Get the workbook and sheet
9          workbook = writer.book
```

```
        worksheet = writer.sheets['Report']

        # Add cell formatting
        from openpyxl.styles import Font, PatternFill, Alignment

        # Header formatting
        header_font = Font(bold=True, color='FFFFFF')
        header_fill = PatternFill(start_color='003864',
                                  end_color='003864',
                                  fill_type='solid')

        for cell in worksheet[1]:  # First row
            cell.font = header_font
            cell.fill = header_fill
            cell.alignment = Alignment(horizontal='center')

        # Column width adjustment
        for column in worksheet.columns:
            max_length = 0
            column_letter = column[0].column_letter
            for cell in column:
                if cell.value:
                    max_length = max(max_length, len(str(cell.value)))
            worksheet.column_dimensions[column_letter].width =
               max_length + 2

# Example usage
report_data = pd.DataFrame({
    'Product': ['Laptop', 'Mouse', 'Keyboard', 'Monitor'],
    'Sales': [45000, 2300, 3400, 28000],
    'Units': [45, 230, 170, 70]
})
create_formatted_report(report_data, 'formatted_report.xlsx')
```

2.4 Converting Between Formats

```
# Convert JSON to Excel
def json_to_excel(json_file, excel_file):
    """Convert JSON data to Excel format."""
    # Load JSON
    with open(json_file, 'r') as f:
        data = json.load(f)

    # Handle different JSON structures
    if isinstance(data, list):
        df = pd.DataFrame(data)
    elif isinstance(data, dict):
        # If dict of lists, treat as multiple sheets
        if all(isinstance(v, list) for v in data.values()):
            with pd.ExcelWriter(excel_file) as writer:
                for key, values in data.items():
                    pd.DataFrame(values).to_excel(writer,
                                                  sheet_name=key,
                                                  index=False)
            return
        else:
            # Single dict to single sheet
```

```
            df = pd.DataFrame([data])

    # Write single DataFrame to Excel
    df.to_excel(excel_file, index=False)

# Convert CSV to Excel with multiple sheets
def csv_files_to_excel(csv_files, excel_file):
    """Combine multiple CSV files into one Excel file."""
    with pd.ExcelWriter(excel_file) as writer:
        for csv_file in csv_files:
            sheet_name = csv_file.replace('.csv', '')
            df = pd.read_csv(csv_file)
            df.to_excel(writer, sheet_name=sheet_name, index=False)
```

3 Part 3: NumPy Array Fundamentals

3.1 Why NumPy Arrays Matter

NumPy arrays are the foundation of scientific computing in Python. They provide dramatic performance improvements over lists for numerical operations. Understanding when and why to use arrays is crucial for data science.

```
import numpy as np
import time

# Performance comparison: Lists vs Arrays
size = 1000000

# List operation
list_data = list(range(size))
start = time.time()
list_result = [x * 2 for x in list_data]
list_time = time.time() - start

# Array operation
array_data = np.arange(size)
start = time.time()
array_result = array_data * 2
array_time = time.time() - start

print(f"List time: {list_time:.4f} seconds")
print(f"Array time: {array_time:.4f} seconds")
print(f"NumPy is {list_time/array_time:.0f}x faster!")
```

3.2 Creating and Using Arrays

```
# Array creation methods
# From lists
scores = np.array([85, 92, 78, 95, 88])

# Initialization functions
zeros = np.zeros(10)                    # 1D array of zeros
ones = np.ones((3, 4))                  # 3x4 array of ones
sequence = np.arange(0, 20, 2)          # [0, 2, 4, ..., 18]
linspace = np.linspace(0, 1, 5)         # 5 evenly spaced values

```

```python
# Random arrays
random_data = np.random.randn(100)      # 100 normal distributed values

# Array attributes
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(f"Shape: {matrix.shape}")         # (2, 3)
print(f"Dimensions: {matrix.ndim}")     # 2
print(f"Size: {matrix.size}")           # 6
print(f"Data type: {matrix.dtype}")     # int64
```

3.3 Essential Array Operations

```python
# Vectorized operations (no loops needed!)
temperatures_f = np.array([72, 68, 75, 71, 78, 82, 79])

# Convert all temperatures to Celsius at once
temperatures_c = (temperatures_f - 32) * 5/9

# Statistical operations
print(f"Mean: {temperatures_c.mean():.1f}")
print(f"Max: {temperatures_c.max():.1f}")
print(f"Min: {temperatures_c.min():.1f}")
print(f"Std Dev: {temperatures_c.std():.1f}")

# Boolean indexing
hot_days = temperatures_f > 75
print(f"Hot days: {temperatures_f[hot_days]}")

# Mathematical functions
values = np.array([1, 4, 9, 16, 25])
sqrt_values = np.sqrt(values)           # Element-wise square root
exp_values = np.exp([0, 1, 2, 3])      # Exponential function
```

3.4 Arrays with pandas Integration

```python
# NumPy and pandas work seamlessly together
# Create DataFrame from arrays
dates = pd.date_range('2025-01-01', periods=7)
temps = np.random.normal(20, 5, 7)
humidity = np.random.uniform(40, 80, 7)

weather_df = pd.DataFrame({
    'date': dates,
    'temperature': temps,
    'humidity': humidity
})

# Use NumPy functions on DataFrame columns
weather_df['temp_normalized'] = (weather_df['temperature'] -
                                 weather_df['temperature'].mean()) /
                                 weather_df['temperature'].std()

# Extract arrays from DataFrame
temp_array = weather_df['temperature'].values  # Returns NumPy array
```

4 Part 4: Data Pipeline Integration

4.1 Building an ETL Pipeline

Combining JSON, Excel, and NumPy for real-world data processing:

```
class DataPipeline:
    """ETL pipeline for multi-format data processing."""

    def __init__(self):
        self.raw_data = None
        self.processed_data = None

    def extract_json(self, json_file):
        """Extract data from JSON source."""
        with open(json_file, 'r') as f:
            self.raw_data = json.load(f)
        print(f"Extracted {len(self.raw_data)} records from JSON")

    def transform_data(self):
        """Transform using NumPy for efficiency."""
        # Convert to DataFrame
        df = pd.DataFrame(self.raw_data)

        # NumPy operations for numerical columns
        if 'values' in df.columns:
            values_array = df['values'].values

            # Efficient array operations
            df['normalized'] = (values_array - values_array.mean()) /
                               values_array.std()
            df['percentile'] = np.percentile(values_array,
                                             np.arange(0, 101, 100/len(df
                                                )))

        self.processed_data = df
        print("Data transformed successfully")

    def load_excel(self, excel_file):
        """Load results to Excel with formatting."""
        with pd.ExcelWriter(excel_file, engine='openpyxl') as writer:
            # Main data sheet
            self.processed_data.to_excel(writer,
                                         sheet_name='Processed Data',
                                         index=False)

            # Summary statistics sheet
            summary = self.processed_data.describe()
            summary.to_excel(writer, sheet_name='Statistics')

            # Add charts sheet with data for visualization
            chart_data = self.processed_data.groupby('category').agg({
                'values': ['mean', 'std', 'count']
            })
            chart_data.to_excel(writer, sheet_name='Chart Data')

        print(f"Results saved to {excel_file}")

# Pipeline usage
```

```
pipeline = DataPipeline()
pipeline.extract_json('sales_data.json')
pipeline.transform_data()
pipeline.load_excel('sales_report.xlsx')
```

4.2 Format Selection Guidelines

Format	Best For	Limitations
JSON	Web APIs, configuration files, NoSQL data, nested structures	No native date types, larger file sizes
Excel	Business reports, formatted data, multiple related tables	Slower for large data, proprietary format
CSV	Simple tabular data, universal compatibility	No data types, single table only
NumPy	Numerical computations, scientific data, performance-critical operations	Homogeneous data only, less flexible

5 Part 5: Practical Application - Student Analytics System

```
class StudentAnalytics:
    """Complete system using JSON config, Excel reports, NumPy analysis."""

    def __init__(self, config_file):
        # Load configuration from JSON
        with open(config_file, 'r') as f:
            self.config = json.load(f)
        self.data = {}

    def load_student_data(self, excel_file):
        """Load student data from Excel."""
        self.data = pd.read_excel(excel_file, sheet_name=None)
        return self

    def analyze_performance(self):
        """Use NumPy for statistical analysis."""
        grades_df = self.data['Grades']

        # Convert grades to NumPy array for fast computation
        grade_values = grades_df['Grade'].values

        # Calculate statistics
        stats = {
            'mean': np.mean(grade_values),
            'median': np.median(grade_values),
            'std': np.std(grade_values),
            'percentiles': {
                '25th': np.percentile(grade_values, 25),
                '75th': np.percentile(grade_values, 75)
            }
        }

        # Grade distribution
        bins = [0, 60, 70, 80, 90, 100]
```

```
        hist, _ = np.histogram(grade_values, bins=bins)
        stats['distribution'] = dict(zip(['F', 'D', 'C', 'B', 'A'],
            hist))

        return stats

    def generate_report(self, output_file):
        """Generate comprehensive Excel report."""
        with pd.ExcelWriter(output_file, engine='openpyxl') as writer:
            # Original data
            for sheet_name, df in self.data.items():
                df.to_excel(writer, sheet_name=sheet_name, index=False)

            # Analysis results
            stats = self.analyze_performance()
            stats_df = pd.DataFrame([stats])
            stats_df.to_excel(writer, sheet_name='Analysis', index=
                False)

            # Student rankings
            grades_df = self.data['Grades']
            rankings = grades_df.groupby('StudentID')['Grade'].agg(['
                mean', 'count'])
            rankings = rankings.sort_values('mean', ascending=False)
            rankings.to_excel(writer, sheet_name='Rankings')

        # Also save summary as JSON
        with open('summary.json', 'w') as f:
            json.dump(stats, f, indent=2)

        print(f"Report generated: {output_file}")

# Usage example
analytics = StudentAnalytics('config.json')
analytics.load_student_data('student_data.xlsx')
analytics.generate_report('student_report.xlsx')
```

Summary and Best Practices

Key Takeaways:

1. **JSON for complex data** - Ideal for nested structures, APIs, and configuration
2. **Excel for business users** - Perfect for reports with formatting and multiple sheets
3. **NumPy for performance** - Essential for numerical computations on large datasets
4. **Format conversion** - pandas bridges all formats seamlessly
5. **Choose wisely** - Each format has specific strengths and use cases
6. **Error handling** - Always validate data when converting between formats
7. **Pipeline thinking** - Combine formats for powerful data processing workflows

Common Pitfalls to Avoid

- Assuming JSON data structure without validation
- Not handling missing or malformed data
- Using lists instead of NumPy arrays for numerical data
- Overwriting Excel files without backup
- Ignoring data types when converting formats
- Not using context managers for file operations

Practice Exercises

Exercise 1: Create a JSON configuration system that loads application settings with validation and default values.

Exercise 2: Build an Excel report generator that reads data from multiple CSV files and creates a formatted multi-sheet workbook.

Exercise 3: Implement a data analysis pipeline that loads JSON data, performs statistical analysis with NumPy, and exports results to Excel.

Exercise 4: Create a format converter that can transform data between JSON, CSV, and Excel formats with proper error handling.

Next Steps and Future Learning

You now have the tools to work with modern data formats professionally:

- **APIs and Web Services**: Use JSON skills for REST API integration
- **Data Science**: NumPy is the foundation for pandas, scikit-learn, and deep learning
- **Business Intelligence**: Create automated Excel reports and dashboards
- **ETL Pipelines**: Build data processing systems for real applications