# ****W8D3 -- Data Processing: Historical Analysis****

JTC Program: Tech Pathways Cohort: S25 Lesson Plan: Historical Data Processing  
Type: Lesson Plan Week / Day: W8D3 Version Date: 05/23/2025

## ****Focus Concepts****

* Understanding how to structure and organize historical data for analysis
* Learning basic data reading and manipulation techniques with Python lists and dictionaries
* Exploring data filtering and searching methods for historical datasets
* Implementing basic statistical analysis on historical information
* Working with CSV files for larger historical datasets
* Building comprehensive analysis systems for historical research

## ****Learning Objectives****

By the end of this session, fellows will be able to:

* Read and display historical data using Python data structures
* Filter and search through historical datasets using various criteria
* Perform basic statistical analysis on historical data (averages, trends, grouping)
* Load and process historical data from CSV files
* Create interactive systems for querying historical information
* Build comprehensive analysis reports combining multiple data processing techniques

## ****Out-of-Scope Objectives****

* Advanced database management and SQL queries
* Complex statistical modeling and hypothesis testing
* Web scraping for historical data collection
* Geographic information systems (GIS) for historical mapping
* Advanced data visualization libraries (beyond basic matplotlib)
* Machine learning applications to historical data

## ****Required Competencies****

* Basic Python programming concepts (variables, loops, functions)
* Understanding of Python data structures (lists, dictionaries)
* Familiarity with basic file operations
* Comfort with basic mathematical concepts (averages, percentages)
* Understanding of conditional logic and iteration

## ****Technical Requirements****

* Python 3.x installed
* Code editor or Jupyter Notebook
* Basic libraries: csv, json, datetime (all part of Python standard library)
* Sample historical datasets (will be provided)

## ****Prerequisites****

* Completion of Python fundamentals
* Understanding of basic programming concepts
* Familiarity with Python data structures
* Basic file handling knowledge

## ****Assigned Reading & Pre-Class Learning****

Estimated Time: 15 minutes

Resources:

* [Python Lists and Dictionaries Tutorial](https://docs.python.org/3/tutorial/datastructures.html) - Review of basic data structures - 8 minutes
* [Working with CSV Files in Python](https://docs.python.org/3/library/csv.html) - Introduction to CSV processing - 7 minutes

## ****Before-Class Mini Quiz Questions (5 questions)****

1. Which Python data structure is best for storing key-value pairs of historical information?
   * A) List
   * \*B) Dictionary
   * C) Tuple
   * D) Set
2. When working with historical data, what is the advantage of using dictionaries over lists?
   * A) Dictionaries are faster to create
   * \*B) Dictionaries allow you to access data by meaningful keys (like "year" or "name")
   * C) Dictionaries use less memory
   * D) Dictionaries are easier to sort
3. What does the csv module in Python help you do?
   * A) Create visual charts
   * \*B) Read and write comma-separated value files
   * C) Connect to databases
   * D) Download data from the internet
4. Which loop structure is most appropriate for processing each item in a list of historical events?
   * A) while loop
   * \*B) for loop
   * C) if statement
   * D) try-except block
5. When filtering historical data, what Python feature allows you to create a new list with only items meeting certain criteria?
   * A) Dictionary comprehension
   * \*B) List comprehension
   * C) Function definition
   * D) Class inheritance

## ****Key Terms****

* **List**: Ordered collection of items that can store multiple pieces of data
* **Dictionary**: Collection of key-value pairs for organized data storage
* **CSV File**: Comma-Separated Values file format for storing tabular data
* **Iteration**: Process of going through each item in a collection one by one
* **Filtering**: Selecting only the data that meets specific criteria
* **Key-Value Pair**: Association between a unique identifier (key) and its corresponding data (value)
* **Data Structure**: Way of organizing and storing data in a program
* **Index**: Position number of an item in a list (starting from 0)
* **String Formatting**: Method of inserting variables into text strings
* **File I/O**: Input/Output operations for reading from and writing to files
* **Object-Oriented Programming**: Programming approach using classes and objects
* **Method**: Function that belongs to a class or object
* **Attribute**: Variable that belongs to a class or object
* **JSON**: JavaScript Object Notation, a format for storing and exchanging data
* **Data Analysis**: Process of examining data to discover patterns and insights

## ****Lesson Schedule & Detailed Script****

### ****6:30 PM -- 6:45 PM: Interactive Check-In****

**Instructor Script:** "Welcome to our lesson on historical data processing! Today we'll learn how to work with historical information using Python. We'll start with simple data structures and progress to building sophisticated analysis systems. By the end of this session, you'll be able to process, analyze, and extract insights from historical datasets. This is a fundamental skill that applies not just to history, but to any field where you need to work with organized information."

**Admin Tasks:**

* Take attendance
* Ensure everyone has Python environment ready
* Check that sample files are accessible

**Prompting Questions:**

* "What types of historical information do you think would be interesting to analyze with code?"
* "Have you ever worked with spreadsheets or data files before? What challenges did you encounter?"

**Poll Questions:**

* "On a scale of 1-5, how comfortable are you with Python lists and dictionaries?"
* "What historical period interests you most: Ancient history, Medieval, or Modern?"

### ****6:45 PM -- 7:05 PM: Session 1 -- Basic Data Reading and Display****

**Objective:** Learn to work with simple historical data using Python lists and dictionaries.

**Instructor Script:** "Let's start by learning how to store and display historical information using Python's built-in data structures. We'll work with lists and dictionaries to organize historical events and see how code can help us manage information systematically."

#### ****Working with Historical Events as Lists and Dictionaries:****

# Basic Historical Data Reading

# Working with simple historical events

# Simple list of historical events with years

historical\_events = [

("Fall of Roman Empire", 476),

("Norman Conquest of England", 1066),

("Black Death begins", 1347),

("Columbus reaches Americas", 1492),

("French Revolution begins", 1789)

]

print("Historical Events:")

print("-" \* 30)

# Loop through and display each event

for event, year in historical\_events:

print(f"{year}: {event}")

**Key Learning Points:**

* Lists can store multiple pieces of related information
* Tuples within lists help organize paired data (event name and year)
* for loops let us process each item systematically
* String formatting (f-strings) makes output readable

#### ****Using Dictionaries for Richer Historical Data:****

# Working with dictionaries for more detailed information

ancient\_cities = {

"Rome": 450000,

"Alexandria": 400000,

"Antioch": 200000,

"Athens": 100000,

"Sparta": 35000

}

print("Ancient City Populations (estimated):")

print("-" \* 35)

# Display city populations

for city, population in ancient\_cities.items():

print(f"{city}: {population:,} people")

# Find the largest city

largest\_city = max(ancient\_cities, key=ancient\_cities.get)

print(f"\nLargest city: {largest\_city} with {ancient\_cities[largest\_city]:,} people")

**Key Learning Points:**

* Dictionaries use meaningful keys instead of numeric indices
* .items() method lets us access both keys and values
* Built-in functions like max() can work with dictionaries
* Number formatting (:,) makes large numbers more readable

#### ****Hands-on Activity (5 minutes):****

Students create their own dictionary of historical figures with birth years and modify the display code.

### ****7:05 PM -- 7:25 PM: Session 2 -- Data Filtering and Searching****

**Objective:** Learn to filter and search through historical datasets using various criteria.

**Instructor Script:** "Now that we can store and display historical data, let's learn how to find specific information within our datasets. Filtering and searching are essential skills for working with any large collection of data."

#### ****Filtering Historical Data by Criteria:****

# Historical battles with detailed information

battles = [

{"name": "Battle of Hastings", "year": 1066, "country": "England", "casualties": 2000},

{"name": "Battle of Gettysburg", "year": 1863, "country": "USA", "casualties": 51000},

{"name": "Battle of Waterloo", "year": 1815, "country": "Belgium", "casualties": 65000},

{"name": "Battle of Stalingrad", "year": 1942, "country": "Russia", "casualties": 2000000},

{"name": "Battle of Marathon", "year": -490, "country": "Greece", "casualties": 6400}

]

# Filter battles by time period

print("Medieval Battles (500-1500 AD):")

print("-" \* 30)

medieval\_battles = []

for battle in battles:

if 500 <= battle['year'] <= 1500:

medieval\_battles.append(battle)

for battle in medieval\_battles:

print(f"• {battle['name']} in {battle['year']}")

**Key Learning Points:**

* Conditional statements (if) help filter data
* Comparison operators can check ranges of values
* Empty lists can be populated based on conditions
* Dictionary keys provide access to specific data fields

#### ****Using List Comprehensions for Efficient Filtering:****

# Find battles with high casualties using list comprehension

major\_battles = [b for b in battles if b['casualties'] > 50000]

print("Major Battles (over 50,000 casualties):")

print("-" \* 40)

for battle in major\_battles:

print(f"• {battle['name']}: {battle['casualties']:,} casualties")

**Key Learning Points:**

* List comprehensions provide a concise way to filter data
* Complex conditions can be applied in a single line
* The resulting list can be immediately used for further processing

#### ****Creating Search Functions:****

def search\_battles\_by\_country(country\_name):

"""Find all battles in a specific country"""

results = []

for battle in battles:

if country\_name.lower() in battle['country'].lower():

results.append(battle)

return results

# Example search

english\_battles = search\_battles\_by\_country("England")

print("Battles in England:")

for battle in english\_battles:

print(f"• {battle['name']} ({battle['year']})")

**Key Learning Points:**

* Functions make search operations reusable
* String methods like .lower() help with case-insensitive searching
* Functions can return lists for further processing

#### ****Hands-on Activity (5 minutes):****

Students write a function to search battles by century and test it with different time periods.

### ****7:25 PM -- 7:40 PM: Session 3 -- Basic Statistical Analysis****

**Objective:** Perform basic statistical analysis on historical data to identify patterns and trends.

**Instructor Script:** "Data becomes more valuable when we can extract insights from it. Let's learn how to calculate statistics and identify patterns in our historical data."

#### ****Calculating Basic Statistics:****

# Roman Emperor data for analysis

roman\_emperors = [

{"name": "Augustus", "start\_year": -27, "end\_year": 14, "dynasty": "Julio-Claudian"},

{"name": "Tiberius", "start\_year": 14, "end\_year": 37, "dynasty": "Julio-Claudian"},

{"name": "Caligula", "start\_year": 37, "end\_year": 41, "dynasty": "Julio-Claudian"},

{"name": "Claudius", "start\_year": 41, "end\_year": 54, "dynasty": "Julio-Claudian"},

{"name": "Nero", "start\_year": 54, "end\_year": 68, "dynasty": "Julio-Claudian"},

{"name": "Vespasian", "start\_year": 69, "end\_year": 79, "dynasty": "Flavian"},

{"name": "Titus", "start\_year": 79, "end\_year": 81, "dynasty": "Flavian"},

{"name": "Domitian", "start\_year": 81, "end\_year": 96, "dynasty": "Flavian"},

{"name": "Trajan", "start\_year": 98, "end\_year": 117, "dynasty": "Nerva-Antonine"},

{"name": "Hadrian", "start\_year": 117, "end\_year": 138, "dynasty": "Nerva-Antonine"}

]

# Calculate reign lengths

reign\_lengths = []

for emperor in roman\_emperors:

length = emperor['end\_year'] - emperor['start\_year']

reign\_lengths.append(length)

print(f"{emperor['name']}: {length} years")

# Calculate statistics

average\_reign = sum(reign\_lengths) / len(reign\_lengths)

shortest\_reign = min(reign\_lengths)

longest\_reign = max(reign\_lengths)

print(f"\nAverage reign: {average\_reign:.1f} years")

print(f"Shortest reign: {shortest\_reign} years")

print(f"Longest reign: {longest\_reign} years")

**Key Learning Points:**

* Mathematical operations can be performed on extracted data
* Built-in functions like sum(), min(), and max() provide quick statistics
* Calculations can reveal patterns not immediately obvious

#### ****Grouping and Analyzing by Categories:****

# Analyze by dynasty

dynasty\_counts = {}

dynasty\_years = {}

for emperor in roman\_emperors:

dynasty = emperor['dynasty']

reign\_length = emperor['end\_year'] - emperor['start\_year']

# Count emperors and total years per dynasty

if dynasty in dynasty\_counts:

dynasty\_counts[dynasty] += 1

dynasty\_years[dynasty] += reign\_length

else:

dynasty\_counts[dynasty] = 1

dynasty\_years[dynasty] = reign\_length

print("Dynasty Analysis:")

for dynasty, count in dynasty\_counts.items():

avg\_reign = dynasty\_years[dynasty] / count

print(f"• {dynasty}: {count} emperors, avg reign {avg\_reign:.1f} years")

**Key Learning Points:**

* Dictionaries can accumulate data by category
* Conditional logic helps build summary statistics
* Average calculations reveal differences between groups

#### ****Hands-on Activity (5 minutes):****

Students calculate statistics for a different historical dataset (e.g., average building heights of ancient monuments).

### ****7:40 PM -- 7:50 PM: Break****

10-minute break

### ****7:50 PM -- 8:20 PM: Session 4 -- Working with CSV Files****

**Objective:** Learn to read, process, and analyze historical data from CSV files.

**Instructor Script:** "Real historical datasets are often stored in files like CSV format. Let's learn how to work with larger datasets and build more sophisticated analysis tools."

#### ****Reading and Processing CSV Data:****

import csv

from datetime import datetime

# Create a sample CSV file with historical trade data

def create\_sample\_data():

"""Create a sample CSV file with historical trade data"""

trade\_data = [

["Year", "Country", "Trade\_Partner", "Export\_Value", "Import\_Value", "Trade\_Type"],

[1850, "Britain", "India", 12500000, 8750000, "Colonial"],

[1850, "Britain", "China", 8200000, 15600000, "International"],

[1850, "France", "Algeria", 3400000, 1200000, "Colonial"],

[1860, "Britain", "India", 15600000, 11200000, "Colonial"],

[1860, "Britain", "China", 9800000, 18900000, "International"],

[1860, "France", "Algeria", 4100000, 1800000, "Colonial"],

[1860, "USA", "Britain", 18500000, 22100000, "International"],

[1870, "Britain", "India", 19200000, 14500000, "Colonial"],

[1870, "Britain", "China", 11400000, 21200000, "International"],

[1870, "France", "Algeria", 5200000, 2400000, "Colonial"],

[1870, "USA", "Britain", 25600000, 31200000, "International"],

[1870, "USA", "France", 8900000, 12300000, "International"]

]

with open('historical\_trade.csv', 'w', newline='', encoding='utf-8') as file:

writer = csv.writer(file)

writer.writerows(trade\_data)

print("Sample CSV file 'historical\_trade.csv' created!")

# Load and process the CSV data

def load\_trade\_data(filename):

"""Load CSV data and convert to usable format"""

data = []

try:

with open(filename, 'r', encoding='utf-8') as file:

reader = csv.DictReader(file)

for row in reader:

# Convert numeric columns

row['Year'] = int(row['Year'])

row['Export\_Value'] = int(row['Export\_Value'])

row['Import\_Value'] = int(row['Import\_Value'])

row['Trade\_Balance'] = row['Export\_Value'] - row['Import\_Value']

data.append(row)

print(f"Loaded {len(data)} trade records")

return data

except FileNotFoundError:

print(f"File {filename} not found. Creating sample data...")

create\_sample\_data()

return load\_trade\_data(filename)

# Load the data

trade\_data = load\_trade\_data('historical\_trade.csv')

**Key Learning Points:**

* CSV files store tabular data in a simple text format
* csv.DictReader automatically creates dictionaries from CSV rows
* Data type conversion is often necessary (strings to numbers)
* Error handling helps manage missing files gracefully

#### ****Building Analysis Classes:****

class HistoricalTradeAnalyzer:

def \_\_init\_\_(self, data):

self.data = data

def analyze\_by\_country(self, country):

"""Analyze trade data for a specific country"""

country\_data = [row for row in self.data if row['Country'] == country]

if not country\_data:

print(f"No data found for {country}")

return

print(f"\n{country} Trade Analysis:")

print("=" \* 40)

# Calculate totals by year

years = {}

for record in country\_data:

year = record['Year']

if year not in years:

years[year] = {'exports': 0, 'imports': 0, 'partners': set()}

years[year]['exports'] += record['Export\_Value']

years[year]['imports'] += record['Import\_Value']

years[year]['partners'].add(record['Trade\_Partner'])

# Display results

for year in sorted(years.keys()):

data = years[year]

balance = data['exports'] - data['imports']

print(f"{year}: Exports: ${data['exports']:,}, Imports: ${data['imports']:,}")

print(f" Balance: ${balance:,}, Partners: {len(data['partners'])}")

# Create analyzer and run analysis

analyzer = HistoricalTradeAnalyzer(trade\_data)

analyzer.analyze\_by\_country("Britain")

analyzer.analyze\_by\_country("USA")

**Key Learning Points:**

* Classes help organize related functions and data
* Methods can perform complex analysis on stored data
* Set data structures automatically handle unique values
* Dictionary accumulation patterns are common in data analysis

#### ****Hands-on Activity (8 minutes):****

Students add a new method to analyze trade by type (Colonial vs International) and compare the patterns.

### ****8:20 PM -- 8:50 PM: Session 5 -- Advanced Analysis System****

**Objective:** Build a comprehensive historical analysis system that combines all previous concepts.

**Instructor Script:** "Now let's bring everything together to create a sophisticated system that can analyze historical timelines, find patterns, and generate comprehensive reports. This represents the kind of system you might build for real historical research."

#### ****Creating a Historical Event Class:****

import json

from datetime import datetime

class HistoricalEvent:

"""Class to represent a historical event with rich metadata"""

def \_\_init\_\_(self, name, year, event\_type, location, significance, participants=None):

self.name = name

self.year = year

self.event\_type = event\_type

self.location = location

self.significance = significance

self.participants = participants or []

def \_\_str\_\_(self):

return f"{self.year}: {self.name} ({self.event\_type})"

def to\_dict(self):

"""Convert event to dictionary for export"""

return {

'name': self.name,

'year': self.year,

'type': self.event\_type,

'location': self.location,

'significance': self.significance,

'participants': self.participants

}

# Create sample events

events = [

HistoricalEvent("Fall of Western Roman Empire", 476, "Political", "Rome", 9,

["Odoacer", "Romulus Augustulus"]),

HistoricalEvent("Norman Conquest of England", 1066, "Military", "England", 9,

["William the Conqueror", "Harold Godwinson"]),

HistoricalEvent("Black Death Begins", 1347, "Social", "Europe", 10,

["Genoese merchants", "European population"]),

HistoricalEvent("Columbus Reaches Americas", 1492, "Exploration", "Caribbean", 10,

["Christopher Columbus", "Spanish Crown"]),

HistoricalEvent("French Revolution Begins", 1789, "Political", "France", 10,

["Third Estate", "Louis XVI"])

]

**Key Learning Points:**

* Classes can model complex real-world entities
* Constructor methods (\_\_init\_\_) set up object properties
* String representation methods (\_\_str\_\_) control how objects display
* Objects can convert themselves to other formats (dictionaries, JSON)

#### ****Building a Comprehensive Timeline Analyzer:****

class HistoricalTimelineAnalyzer:

def \_\_init\_\_(self):

self.events = events

def analyze\_by\_century(self):

"""Analyze events by century"""

centuries = {}

for event in self.events:

century = (event.year // 100) + 1

if century not in centuries:

centuries[century] = []

centuries[century].append(event)

for century in sorted(centuries.keys()):

events\_list = centuries[century]

print(f"\n{century}th Century ({len(events\_list)} events):")

print("-" \* 30)

for event in sorted(events\_list, key=lambda x: x.year):

print(f" {event.year}: {event.name} (Significance: {event.significance}/10)")

def analyze\_by\_type(self):

"""Analyze events by type"""

type\_analysis = {}

for event in self.events:

event\_type = event.event\_type

if event\_type not in type\_analysis:

type\_analysis[event\_type] = {'count': 0, 'total\_significance': 0, 'events': []}

type\_analysis[event\_type]['count'] += 1

type\_analysis[event\_type]['total\_significance'] += event.significance

type\_analysis[event\_type]['events'].append(event)

print("\nAnalysis by Event Type:")

print("=" \* 30)

for event\_type, data in type\_analysis.items():

avg\_significance = data['total\_significance'] / data['count']

print(f"{event\_type}: {data['count']} events, avg significance {avg\_significance:.1f}")

def interactive\_query\_system(self):

"""Interactive system for querying historical data"""

print("\nInteractive Historical Query System")

print("Available commands:")

print("1. 'year YYYY' - Find events in specific year")

print("2. 'type EVENT\_TYPE' - Find events of specific type")

print("3. 'quit' - Exit query system")

while True:

query = input("\nEnter query: ").strip().lower()

if query == 'quit':

break

parts = query.split()

if len(parts) >= 2:

command = parts[0]

parameter = ' '.join(parts[1:])

if command == 'year':

year = int(parameter)

results = [e for e in self.events if e.year == year]

if results:

for event in results:

print(f" • {event}")

else:

print(f" No events found for year {year}")

elif command == 'type':

results = [e for e in self.events if parameter.lower() in e.event\_type.lower()]

if results:

for event in results:

print(f" • {event}")

else:

print(f" No {parameter} events found")

# Create and use the analyzer

analyzer = HistoricalTimelineAnalyzer()

analyzer.analyze\_by\_century()

analyzer.analyze\_by\_type()

**Key Learning Points:**

* Complex analysis can be broken into focused methods
* Data grouping patterns are fundamental to analysis
* Interactive systems can make data exploration more engaging
* User input validation and error handling improve usability

#### ****Exporting Analysis Results:****

def export\_analysis(self, filename="historical\_analysis.json"):

"""Export analysis results to JSON file"""

export\_data = {

'metadata': {

'total\_events': len(self.events),

'analysis\_date': datetime.now().strftime("%Y-%m-%d %H:%M:%S")

},

'events': [event.to\_dict() for event in self.events]

}

with open(filename, 'w', encoding='utf-8') as file:

json.dump(export\_data, file, indent=2, ensure\_ascii=False)

print(f"Analysis exported to {filename}")

# Add this method to the HistoricalTimelineAnalyzer class

HistoricalTimelineAnalyzer.export\_analysis = export\_analysis

**Key Learning Points:**

* JSON format is excellent for data interchange
* Metadata helps document when and how analysis was performed
* File export makes results available to other programs
* Method addition demonstrates Python's flexibility

#### ****Hands-on Activity (10 minutes):****

Students add a new method to find events within a date range and export their results to JSON.

### ****8:50 PM -- 9:20 PM: Capstone Work Session****

**Activity:** Work on capstone project, applying historical data processing concepts.

### ****9:20 PM -- 9:30 PM: Wrap-Up & Final Questions****

**Instructor Script:** "Today we've built a complete foundation for processing historical data with Python. We started with simple lists and dictionaries, learned to filter and search data, performed statistical analysis, worked with CSV files, and created sophisticated analysis systems. These skills transfer to any domain where you need to process and analyze structured information."

**Review Key Points:**

* Python data structures provide flexible ways to organize information
* Filtering and searching help extract relevant subsets of data
* Statistical analysis reveals patterns and trends
* CSV files enable work with larger datasets
* Object-oriented programming helps organize complex analysis systems
* Interactive systems make data exploration more engaging

**Prompting Question:** "What type of historical analysis would you most like to build using these techniques?"

## ****After-Class Quiz (5 questions)****

1. What is the main advantage of using dictionaries instead of lists for storing historical data?
   * A) Dictionaries are faster to create
   * \*B) Dictionaries let you access data using meaningful names instead of numeric positions
   * C) Dictionaries use less computer memory
   * D) Dictionaries can only store numbers
2. When filtering a list of historical events, which Python feature provides the most concise syntax?
   * A) for loops with if statements
   * \*B) List comprehensions
   * C) while loops
   * D) Dictionary methods
3. What does the csv.DictReader function do?
   * A) Creates dictionaries from scratch
   * B) Converts dictionaries to CSV format
   * \*C) Reads CSV files and converts each row to a dictionary
   * D) Counts the number of rows in a CSV file
4. In object-oriented programming, what is a method?
   * A) A type of variable
   * B) A way to store data
   * \*C) A function that belongs to a class
   * D) A file format
5. Why might you export analysis results to JSON format?
   * A) JSON files are smaller than other formats
   * B) JSON files can only be read by Python
   * \*C) JSON format is widely supported and human-readable
   * D) JSON files are automatically backed up

## ****Assignment (Optional Extension)****

Create a historical analysis system for a topic of your choice:

1. Define at least 10 historical events with multiple attributes
2. Implement filtering by at least two different criteria
3. Calculate at least three different statistics
4. Create an interactive query system
5. Export results to JSON format

Example topics: Ancient civilizations, Scientific discoveries, Artistic movements, Technological innovations, Political revolutions

**Deliverables:**

* Python script with complete analysis system
* Sample data file (CSV or JSON)
* Brief report explaining your findings
* Screenshot of interactive query session