Introduction to Python

Fondren Library
Research Data Services

Why Multiple Files?

Reasons Why We Use Multiple Files in Python

1. **Modularity:**
   a. Breaking down data into multiple files keeps related data organized and manageable.
   b. Facilitates maintenance and readability of data.
2. **Separation of Concerns:**
   a. Different files can store different types of data, e.g., configurations, logs, and records.
   b. Helps keep data logically separated and easier to handle.
3. **Scalability:**
   a. Managing large amounts of data is more efficient when divided into multiple files.
   b. Prevents single files from becoming too large and unwieldy.
4. **Concurrent Processing:**
   a. Allows simultaneous reading/writing operations on different files.
   b. Improves performance and efficiency in multi-threaded or distributed systems.
5. **Backup and Recovery:**
   a. Easier to backup and restore smaller, individual files.
   b. Reduces risk of data loss and simplifies recovery processes.
6. **Data Aggregation:**
   a. Enables combining data from various sources into a cohesive dataset.
   b. Facilitates data analysis and reporting by aggregating smaller datasets.
7. **Practical Example:**
   a. Storing user data, transaction logs, and configuration settings in separate files for an application.
   b. Example:
      i. `users.txt` for user data.
      ii. `transactions.txt` for transaction logs.
      iii. `config.txt` for configuration settings.

Reasons Why We Use Multiple Files in Python

Data Cleaning Project:

Raw data in raw_data.csv

Cleaning scripts in data_cleaning.py

Cleaned data in cleaned_data.csv

Cleaning logs in cleaning_report.txt

Web Scraping Project:

Scraping script for site1 in scrape_site1.py

Scraping script for site2 in scrape_site2.py

Data from site1 in site1_data.json

Data from site2 in site2_data.json

Data Analysis Project:

Data loading and preprocessing in data_preprocessing.py

Model training script in model_training.py

Training data in train.csv

Test data in test.csv

Model results in results.pkl

Data and Results:

Raw data in data/raw/

Cleaned data in data/cleaned/

Feature data in data/features/

Model results and logs in results/

Data Preparation:

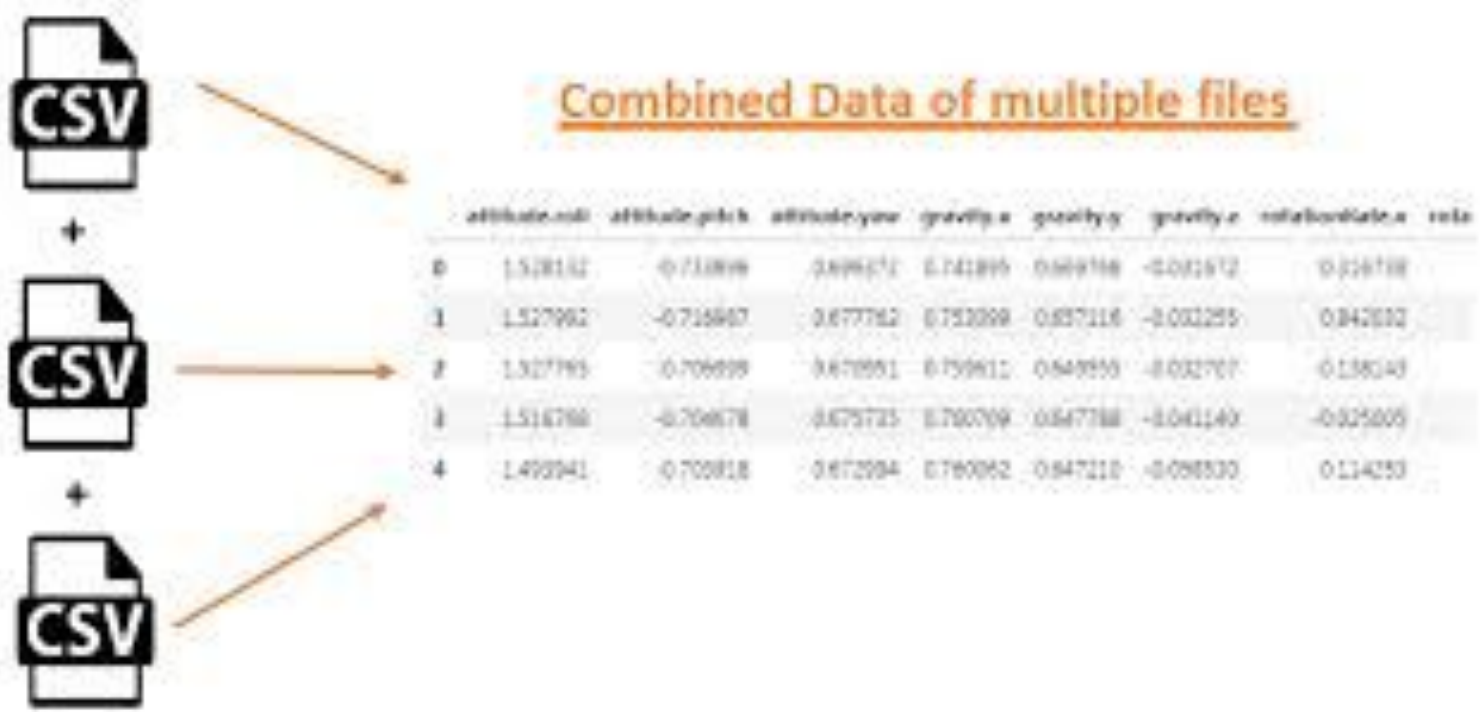
data_loader.py: Script to load raw data from various sources (e.g., CSV, SQL database).

data_cleaning.py: Script to clean and preprocess the data (e.g., handling missing values, encoding categorical variables).

feature_engineering.py: Script to create new features based on the existing data (e.g., calculating customer tenure, average purchase value).

`Machine Learning`
`train_model.py`: Script to train different machine learning models (e.g., decision trees, logistic regression) on the prepared data.
`evaluate_model.py`: Script to evaluate the performance of trained models using metrics like accuracy, precision, recall, and F1 score.

Reasons Why We Use Multiple Files in Python



File Handling

Opening and closing files

Slide 2: Opening and Closing Files in Python

- **Opening a File:**
  - Use the `open()` function to open a file.
  - Syntax: `file = open('filename', 'mode')`
- **File Modes:**
  - `'r'`: Read mode. Opens a file for reading.
  - `'w'`: Write mode. Opens a file for writing (creates a new file or truncates an existing file).
  - `'a'`: Append mode. Opens a file for appending (creates a new file if it does not exist).

Example:

```
file = open('example.txt', 'r')  # Opens the file in read mode
```

- **Closing a File:**
  - Syntax: `file.close()`

Example:

```
file = open('example.txt', 'r')
content = file.read()
file.close()
```

Shortcut: Using the context manager

For most basic read, write, and edit operations, you can use the python context manager! It automatically handles the closing and file handling after your code has executed, so you don't have to. This is the most common way to edit files, but does not work for some more complex operations.

Syntax:

```
with open('example.txt', 'r') as file:
    content = file.read()
```

Explanation:

When a file is opened with with open('example.txt', 'r') as file:, the file object is created and the code inside the with block is executed.

When the block is exited, similarly to how variables created in a for loop only exist within the for loop, the context manager automatically calls file.close(), ensuring the file is properly closed.

It also handles closing the file even if there are any errors!

Example:

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
# No need to explicitly close the file
```

Reading Multiple Files

Example file: properties.txt

```
ID,Name,Location,Price
1,Green Villa,New York,750000
2,Blue Cottage,Los Angeles,550000
3,Red House,Chicago,620000
4,White Mansion,Houston,1200000
```

Reading the entire file

Use case: When you need to process the entire content of the file as a single string.

Syntax:

```
file = open('properties.txt', 'r')  # Open the file in read mode
content = file.read()               # Read the entire content
print(content)                      # Print the content
file.close()                        # Close the file
```

This will print the entire content of the file, and it will save the text of the file in the variable content, as a string.

Reading the file line by line

Use case: When you need to process each line individually, e.g., filtering specific lines.

Syntax:

```
file = open('properties.txt', 'r')  # Open the file in read mode
line = file.readline()              # Read the first line
index = 0                           # Initialize line index

while line:                         # Loop until there are no more lines
    if index % 2 == 0:              # Print every other line
        print(line, end='')         # Print the line
    line = file.readline()          # Read the next line
    index += 1                      # Increment line index

file.close()                        # Close the file
```

Output:

```
ID,Name,Location,Price
2,Blue Cottage,Los Angeles,550000
4,White Mansion,Houston,1200000
```

Reading each line of the file into a list

Use case: When you need to work with all lines as a list, e.g., extracting specific columns.

Syntax:

```
file = open('properties.txt', 'r')  # Open the file in read mode
lines = file.readlines()            # Read all lines into a list
file.close()                        # Close the file

state_names = []                    # Initialize an empty list for state names

for line in lines[1:]:              # Skip the header and process each line
    columns = line.split(',')       # Split the line into columns
    state_names.append(columns[2])  # Append the state name (3rd column)

print(state_names)                  # Print the list of state names
```

Output:

```
['New York', 'Los Angeles', 'Chicago', 'Houston']
```

Writing to Multiple Files

Writing Data to Different Types of Files

The file extension (e.g., .txt, .csv, .json) indicates the type of data the file contains. It helps the operating system and applications recognize how to handle the file.

Text Files (.txt):

- Store plain text data.
- Useful for storing simple, unstructured data like notes, logs, and lists.

Example Code:

```
with open('data.txt', 'w') as file:
    file.write('Hello, World!\nThis is a text file.')
```

CSV Files (.csv):

- Store tabular data in a comma-separated values format.
- Commonly used for data exchange between databases.

```
import csv

data = [
    ['ID', 'Name', 'Location', 'Price'],
    [1, 'Green Villa', 'New York', 750000],
    [2, 'Blue Cottage', 'Los Angeles', 550000]
]
```

Example Code:

```
with open('data.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)
```

JSON Files (.json):

- Store data in a structured, human-readable format.
- Ideal for storing complex data structures like dictionaries.

```
import json

data = {
    'ID': 1,
    'Name': 'Green Villa',
    'Location': 'New York',
    'Price': 750000
}
```

Example Code:

```
with open('data.json', 'w') as file:
    json.dump(data, file, indent=4)
```

Writing to Files

Step 1: Create Data:

```
new_properties = [
    "ID,Name,Location,Price\n",
    "5,Black Apartment,Boston,300000\n",
    "6,Yellow Bungalow,San Francisco,900000\n"
]
```

Write mode!

Step 2: Write data into file

```
file = open('new_properties.txt', 'w')  # Open the file in write mode
file.writelines(new_properties)         # Write the list of strings to the file
file.close()                            # Close the file
```

IMPORTANT: If the file does not exist, it will create a new file with that name, and write the data into that file. If the file exists, **it will overwrite any data** inside of the file. To add data to a file, you have to use appending.

Appending to Files

Step 1: Create Data:

```
new_properties = [
    "ID,Name,Location,Price\n",
    "5,Black Apartment,Boston,300000\n",
    "6,Yellow Bungalow,San Francisco,900000\n"
]
```

Append mode!

Step 2: Append data into file

```
file = open('properties.txt', 'a')  # Open the file in append mode
file.writelines(more_properties)    # Append the list of strings to the file
file.close()                        # Close the file
```

By changing this from a **w**, to an **a,** we enter append mode, and can now add data to **the end of the file**, instead of overwriting the data in the file. Don’t worry, if the file does not exist, it will still create a new file.

GitHub Repos

What Is GitHub?

GitHub is a web-based platform used for version control and collaborative software development. It allows multiple developers to work on projects simultaneously, tracking changes, managing versions, and collaborating efficiently. GitHub utilizes Git, a distributed version control system, to handle project versions, making it easy to revert to previous states, merge changes, and handle conflicts. It also provides features like issue tracking, project management, and integration with other tools, making it a central hub for open-source and private software development projects.

Version Control: Keeps track of every change made to the codebase, allowing you to revert to previous versions if necessary.

Collaboration: Facilitates teamwork by allowing multiple developers to contribute to the same project, manage code reviews, and merge changes seamlessly.

Project Management: Offers tools like issue tracking, milestones, and project boards to help manage development tasks and track progress.

Integration: Integrates with various tools and services, such as continuous integration/continuous deployment (CI/CD) pipelines, to automate workflows.

Connecting a GitHub Repository to Visual Studio Code

1. **Install Git:**
   a. Ensure Git is installed on your system.
   b. https://git-scm.com/downloads
2. **Install GitHub Extension:**
   a. Install the GitHub extension for VS Code.
   b. Go to Extensions (Ctrl+Shift+X), search for "GitHub", and install the extension.
3. **Clone your GitHub repository to your local machine.**
   a. Open VS Code.
   b. Go to the Source Control view (Cmd/ctl+shift+P)
   c. Click on "Git Clone".
   d. Enter your repository URL and select a local folder to clone the repository.
4. **Open the Repository:**
   a. Open the cloned repository in VS Code.
   b. File -> Open Folder -> Select the cloned repository folder.
5. **Configure Git in VS Code:**
   a. Configure Git settings in VS Code if needed.
   b. Go to File -> Preferences -> Settings and search for "Git".

6. **Make Changes and Commit:**

   a. Edit files, stage changes, and commit.
   b. Make changes to your files.
   c. Go to Source Control view.
   d. Stage changes by clicking the "+" icon next to the files.
   e. Enter a commit message and click the checkmark to commit.

7. **Push your local commits to GitHub.**

   a. Go to Source Control view.
   b. Click on the "..." menu and select "Push".

8. **Pull Changes from GitHub:**

   a. Go to Source Control view.
   b. Click on the "..." menu and select "Pull".

Activity: Finding GitHub Repositories