PYTHON: FILE HANDLING

]

]

]

]

File Handling: Contents

- ✔ What if File Handling?
- ✔ Types of Files
- ✔ Different File modes
- ✔ Opening, Reading, Writing File
- ✔ Reading a File

PYTHON: File Handling

Python: What is File Handling?

- **File Handling** is how a program:
  - **creates**, **opens**, **reads**, **writes**, **updates**, **deletes**, and **closes**
  - files stored on a computer (hard disk). perform an operation.
- Files store data permanently on the computer.
- File handling allows you to **store**, **access**, and **manipulate data** outside your program.

Python: Real World Analogy

- Think of file handling like a **library**.
- You have to walk to the shelf,
  1. **pick a book**,
  2. **read or write in it**,
  3. and then **put it back**.
- If you don't put it back (**close it**), the next person can't find it, or the book might get damaged.

Python: Real World Analogy

- Working with files in Python is like using a **physical filing cabinet**.
- You have to walk over to it,
  - ✓ **open a drawer**,
  - ✓ **read or write on a paper**, and—most importantly—
  - ✓ **close the drawer** when you're done so nothing gets lost or damaged.
- In Python, we follow the same three-step pattern:
  - ✓ **Open → Work → Close**.

Why File Handling is Important

1. Store user data
2. Save program output
3. Read configuration files
4. Build real applications (student systems, logs, reports)

Python: Types of Files

- In your journey as a developer, you'll run into different file types.
- They can be broadly split into **Text** *(human-readable)* and **Binary** *(computer-readable)*.
- **'.txt'** → **Plain Text**: Simple *notes*, *logs*, and *README* files.
- **'.csv'** → **Data:** Tabular *data*, *spreadsheets*, and *databases*.
- **'.json'** → **Data/Web:** Web APIs, configuration settings, and complex

Python: Types of Files

- **‘.xml’ → Data/Web:** Older way of sending web data (looks like HTML).
- **‘.yaml’ → Config**: Configuration for servers and "DevOps" tools.
- **‘.jpg’ / ‘.png’ → Binary:** Image files (requires libraries like **Pillow** to open).
- **‘.pdf’ → Binary:** Document files (requires libraries like **PyPDF2**).
- **‘.py’ → Code:** Your Python scripts!

Python: Types of Files - Others

1. Text files → **.log, .yml, .md**
2. Binary files → **images**, **audio**, **videos** (.jpg, .mp3, .exe)

👉 We are mostly going to be using text files.

Python: File Opening

- In Python, we use the **open()** function
- The **open()** function takes two parameters; **filename**, and **mode**.
- **Syntax:**

```
open(filename, mode)
```

- **Example:**

```
file = open("data.txt", "mode")
```

The File Opening Modes

- When you open a file using the **open() function**, you must tell Python how you intend to use it.
- If you don't specify, it defaults to **'r'** *(read)*.
- **‘r’ → Read** (default): Opens for reading.
  - Errors if the file doesn't exist.
- **‘w’ → Write:** Opens for writing.
  - **Deletes** everything in the file first!

The File Opening Modes

- **‘a’ → Append:** Opens for writing.
  - Adds new text to the **end of the file**.
- **‘x’ → Create:** Creates a new file.
  - Errors if the file already exists.
- **‘rb’ / ‘wb’ → Binary:** Used for non-text files like images or PDFs.
  - Errors if the file already exists.

The File Opening Modes

Mode	Name	Description
'r'	Read	Opens for reading. Errors if the file doesn't exist.
'w'	Write	Opens for writing. **Deletes everything** in the file first!
'a'	Append	Opens for writing. Adds new text to the **end** of the file.
'x'	Create	Creates a new file. Errors if the file already exists.
'rb' / 'wb'	Binary	Used for non-text files like images or PDFs.

Python: Reading Files

- There are three main ways to **pull data out of a file**.
- Choose the one that fits your data size.
- **read():** Grabs the entire file as one giant string.
  - ✓ *Be careful with huge files*!
- **readline():** Grabs just one line at a time.
- **readlines():** Puts every line into a list of strings.

Python: Reading Files – read()

- Read entire file

```
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

Python: Reading Files – readline()

- Read file line by line

```
file = open("data.txt", "r")

print(file.readline())

file.close()
```

Python: *readline()* Method

- While **read()** grabs the whole file at once, **readline()** reads **one line at a time.**
- This is great for memory if you have a massive file (like a 10GB log file).

```
with open("test.txt", "r") as f:
    line1 = f.readline() # Reads the first line
    line2 = f.readline() # Reads the second line
```

Python: Reading Files – readlines()

- With the **readlines()** method, the file will be read and its content returned in a list with **each line** being a **list item**

```
file = open("data.txt", "r")
lines = file.readlines()
print(lines)
file.close()
```

Python: Loop through a File

- This means reading the file **contents line by line** using a **loop** instead of loading the entire file at once.

```
file = open("data.txt", "r")

for line in file:
    print(line.strip())

file.close()
```

- The **for *line* in *file*** statement automatically reads *each line one at a time*.
- You can process each line (e.g., ***count words***, ***search for keywords***) inside the loop.

Python: Loop through a File

- **Using while loop:**

```
file = open("data.txt", "r")

while True:
    line = file.readline()
    if not line:
        break
    print(line.strip())

file.close()
```

Python: Writing to Files (*w* mode)

- **Warning: w** deletes existing content.

```
file = open("data.txt", "w")
file.write("Hello Python\n")
file.write("File Handling Example")
file.close()
```

Python: Writing to Files (*w* mode)

- **Practical Example:** Storing Students Data.

```
file = open("students.txt", "w")
file.write("Daniel\n")
file.write("Sarah\n")
file.write("John\n")
file.close()
```

Python: Appending Data (*a* mode)

- If you use **'w',** your **previous work** is **deleted**.
- To keep what you have and just add more**, use 'a'.**

```
file = open("students.txt", "a")
file.write("\nMary")
file.close()
```

👉 Adds data without **deleting** existing content.

Python: Creating New File (*x* mode)

- ⚠**Warning:** ***x mode*** will give an error if *file already exists*.

```
file = open("newfile.txt", "x")

file.write("New file created")

file.close()
```

Python

Create, Open, Append, Read, Write

- **Modes: *w, a, r***

```
# WRITING (and creating)
with open("test.txt", "w") as f:
    f.write("Hello Python!\n")

# APPENDING
with open("test.txt", "a") as f:
    f.write("Adding a new line.")

# READING
with open("test.txt", "r") as f:
    print(f.read())
```

Check if File or Directory Exists

- Before you try to **open a file**, it's polite to check if it's actually there.
- To **check if it exists**, you need to import the **os module**.

Check if File Exists

- File exists?

```
import os


if os.path.exists("data.txt"):
    print("File exists")
else:
    print("File does not exist")
```

Check if Directory Exists

- Directory exists?

```
if os.path.isdir("myfolder"):
    print("Directory exists")
```

- Check if Path is a File:

```
if os.path.isfile("data.txt"):
    print("It is a file")
```

Check if File or Directory Exists

```python
import os

if os.path.exists("test.txt"):
    print("File exists!")
else:
    print("File not found.")

# Specifically for folders/directories
if os.path.isdir("my_folder"):
    print("That is a folder.")
```

Deleting and Checking Files

- To **delete a file**, you need to import the **os module**.

```
import os

# Check if file exists
if os.path.exists("notes.txt"):
    print("File found!")
    # os.remove("notes.txt") # Uncomment to delete
else:
    print("The file does not exist.")
```

Python: Using *with* Statement

- In the previous examples, you had to manually call **.close().**
- If your codes were to *crashed* before that line, the **file stayed "locked".**
- Using the ***with statement*** (a Context Manager), it automatically closes the file for you, even if an error occurs.
- The with statement acts like an ***automatic assistant*** that closes the file for you when you're finished.

Python: *with* Statement

- **Reading:**

```
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
```

- **Append:**

```
with open("notes.txt", "a") as file:
    file.write("\nThis line is appended to the end.")
```

✔ No **close()** function needed now.

Python: Count Words in File

```
with open("data.txt", "r") as file:
    text = file.read()
    words = text.split()
    print("Total words:", len(words))
```

Python

Python: Copy File Method *(shutil)*

- To copy files, we use the **shutil** (*shell utilities*) module.
- **shutil.copy(src, dst):**
  - Copies the file content and permissions.
- **shutil.copystat(src, dst):**
  - Does not copy the content.
  - It only copies the ***"metadata"*** (the time it was created, permissions, etc.) from one file to another.

Python: Copy File Method *(shutil)*

- **shutil.copy():** Copies file content only

```
import shutil


shutil.copy("source.txt", "destination.txt")
```

```
import shutil


shutil.copy("test.txt", "test_backup.txt")
```

Python: Copy File Method *(shutil)*

- **shutil.copystat():** Copies file metadata (permissions, timestamps).

```
shutil.copystat("source.txt", "destination.txt")
```

- **Shutil.copy2():** Copies both the file contents and metadata

```
shutil.copy2("source.txt", "destination.txt")
```

Python

Rename Files and Directories

- The **os.rename()** method from the **os module** is like the ***"Right Click > Rename"*** feature on your computer.

```
import os


os.rename("old.txt", "new.txt")
```

Python

Rename Files and Directories

- **Rename a File:**

```
import os


os.rename("old.txt", "new.txt")
```

- **Rename a Directory:**

```
os.rename("old_folder", "new_folder")
```

- ⚠ Error if **file** or **folder** does not exist

Python

Rename Files and Directories

```python
import os


# Renaming a file
os.rename("test_backup.txt", "final_version.txt")


# Renaming a folder
# os.rename("old_folder", "new_folder")
```

Python: ZIP File Handling

- Python can **"zip"** and **"unzip"** folders using the ***zipfile module***.
- Think of this as putting your files into a **vacuum-sealed bag** to save space.
- **Create a ZIP File:**

```
import zipfile


with zipfile.ZipFile("files.zip", "w") as zip:
    zip.write("data.txt")
    zip.write("test.txt")
```

Python: ZIP File Handling

- **Create a ZIP File:**

```
with zipfile.ZipFile("files.zip", "r") as zip:
    zip.extractall("output_folder")
```

- **List ZIP Contents:**

```
with zipfile.ZipFile("files.zip", "r") as zip:
    print(zip.namelist())
```

Python: ZIP File Handling

```python
import zipfile

# Creating a ZIP
with zipfile.ZipFile("my_archive.zip", "w") as myzip:
    myzip.write("test.txt")

# Extracting a ZIP
# with zipfile.ZipFile("my_archive.zip", "r") as myzip:
#     myzip.extractall("extracted_folder")
```

Exception Handling: Try, Except, Finally

- Files are *notorious* for causing ***crashes*** (e.g., you try to read a file that isn't there).
- Always wrap your file logic (code) in a ***try...except block,*** the **safety net**, in order to keep your program from dying.
- If something goes wrong (like a file is missing), your program won't crash; it will handle the error gracefully.
- In other languages (like Java), you might hear **"Try, Catch."** In Python, we say **try** and **except**.

Exception Handling: Try, Except, Finally

- **Basic Syntax:**

```
try:
    # risky code
except:
    # error handling
finally:
    # always runs
```

Exception Handling: Try, Except, Finally

- **The Components:**

1. **try:** The code you want to run.
2. **except:** The code that runs if an error happens.
3. **else (Optional):** Runs only if no error happened.
4. **finally (Optional):** Runs no matter what (even if there was an error).
    - Great for cleanup.

Python

Exception Handling: Try, Except, Finally

- **Example:** File Not Found

```
try:
    file = open("data.txt", "r")
    print(file.read())
except FileNotFoundError:
    print("File not found!")
finally:
    print("Program ended")
```

Exception Handling: Try, Except, Finally

- **Example:** Multiple Exceptions

```
try:
    x = int(input("Enter number: "))
    print(10 / x)
except ValueError:
    print("Invalid number")
except ZeroDivisionError:
    print("Cannot divide by zero")
finally:
    print("Done")
```

Exception Handling: Example

```
try:
    with open("missing_file.txt", "r") as f:
        print(f.read())
except FileNotFoundError:
    print("Error: I couldn't find that file!")
except Exception as e:
    print(f"Something else went wrong: {e}")
finally:
    print("The operation is finished.")
```

CSV Files (Spreadsheets in Text)

- CSV stands for **Comma Separated Values**.
  - ✓ It is basically an ***Excel sheet*** but saved as a simple text file.
  - ✓ Each ***line is a row***, and each ***comma*** represents a **new column**.
- **How to Read a CSV:**
  - ✓ Python has a ***built-in*** **csv module** that makes this very easy.

CSV Files (Spreadsheets in Text)

- **Example:** *student.csv*

```
id,name,age
1,Daniel,22
2,Mary,21
3,John,23
```

- Each line = one record
- Each comma = separates values

Why CSV Files are Used

- ✓ Store ***table-like*** data
- ✓ Easy to read
- ✓ Used in Excel, databases, data science

Python

Python CSV Module

- Python has a *built-in module* called **csv**.

```
import csv
```

- It is the Python library used to **read from** and **write to CSV**
- The module makes it easy to handle *structured data* without needing ***external libraries***.

Python: Reading a CSV File

```
import csv


with open("students.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Python: Reading a CSV File

- **Output:**

```
['id', 'name', 'age']
['1', 'Daniel', '22']
['2', 'Mary', '21']
['3', 'John', '23']
```

Python: Reading a CSV File

```python
import csv


# Reading a CSV
with open("employees.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        # Each 'row' is a list: ['Name', 'Role', 'Salary']
        print(f"Name: {row[0]}, Job: {row[1]}")
```

Python: Reading CSV as Dictionary

```
import csv


with open("students.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(row)
```

Python: Reading CSV as Dictionary

- **Output:**

```
{'id': '1', 'name': 'Daniel', 'age': '22'}
{'id': '2', 'name': 'Mary', 'age': '21'}
```

Python: Writing to a CSV File

```python
import csv


with open("employees.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["id", "name", "salary"])
    writer.writerow([1, "Alice", 5000])
    writer.writerow([2, "Bob", 4500])
```

Python: Writing to a CSV File

```
with open("output.csv", "w", newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Score"]) # Header
    writer.writerow(["Alice", 95])     # Data
```

Python: Appending to CSV

```
import csv


with open("employees.csv", "a", newline="") as file:
    writer = csv.writer(file)
    writer.writerow([3, "John", 6000])
```

JSON Files (The Language of the Web)

- JSON stands for **JavaScript Object Notation**.
  - ✓ It is the most common way data is sent over the internet.
  - ✓ Python, JSON looks almost exactly like a **Dictionary**.
  - ✓ It stores data as **key–value pairs**.
- JSON is a **lightweight format** for **storing** and exchanging **structured data**, often used in ***web applications*** and ***APIs***.

JSON Files

- **Example:** *data.json*

```
{
    "name": "Daniel",
    "age": 22,
    "course": "Computer Science"
}
```

Why JSON is Used

- ✓ Used in APIs
- ✓ Used in web & mobile apps
- ✓ Easy to read
- ✓ Stores complex data

Python: Writing JSON to a File

```python
import json

student = {
    "id": 1,
    "name": "Daniel",
    "age": 22,
    "courses": ["Python", "Databases"]
}

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

- ✔ **indent=4** makes it readable

Python: Reading JSON from a File

```python
import json


with open("student.json", "r") as file:
    data = json.load(file)


print(data)
print(data["name"])
```

Python: Writing JSON to a File

```python
# 1. Writing a Dictionary to a JSON file
user_data = {
    "name": "Bob",
    "age": 30,
    "skills": ["Python", "Cooking"]
}

with open("user.json", "w") as file:
    json.dump(user_data, file, indent=4) # indent makes it pretty to read

# 2. Reading from a JSON file
with open("user.json", "r") as file:
    data = json.load(file)
    print(data["name"]) # Output: Bob
```

JSON with Multiple Records (List of Dictionaries)

```
students = [
    {"id": 1, "name": "Daniel"},
    {"id": 2, "name": "Mary"},
    {"id": 3, "name": "John"}
]


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

Reading It Back (Multiple Records)

```
with open("students.json", "r") as file:
    students = json.load(file)


for s in students:
    print(s["name"])
```

Practical Real-World Workflow

- A common task for a ***Python programmer*** looks like this:
  - ✓ **Read** a raw **.txt** file containing logs.
  - ✓ **Process** the data using **Functions** and **Loops**.
  - ✓ **Save** the cleaned results into a **.csv** so a manager can open it in Excel.
  - ✓ **Upload** a summary to a web server as a **.json** file.

A Witty Summary

- File handling is about being a responsible digital citizen:
  - ✓ **Check** if it's there,
  - ✓ **Open** it correctly,
  - ✓ **Handle** errors if they happen, and always –
  - ✓ **Close** your resources (or let with do it for you).

Python: File Handling

Python