Unit 5 – Exception Handling and File Operations

Exceptions in Python handle **errors gracefully** so programs don't crash unexpectedly. File operations allow us to **store, retrieve, and process data** from files.

Subtopic 5.1: Python Exception Hierarchy

1. Definition / Concept

- An **exception** is an **error detected during program execution**.
- Python provides a **hierarchy of exception classes** (all inherit from BaseException).
- Common ones:
  - Exception → Base class for most errors.
  - ZeroDivisionError, TypeError, ValueError, FileNotFoundError, etc.

📌 **Key Point**: Exception hierarchy allows us to **catch errors at different levels** (specific vs general).

2. Analogy / Real-Life Connection

Imagine you're **filling an online form**:

- Wrong password → specific error (like ValueError).
- Network down → another specific error (like ConnectionError).
- But all are still "problems" under one **general error** category.

3. Syntax

```
try:
    # risky code
except SpecificError:
    # handle specific error
except Exception:
    # handle general errors
```

4. Step-by-Step Explanation

1. All exceptions in Python inherit from **BaseException**.
2. Two main branches:
   - SystemExit, KeyboardInterrupt (not usually caught).

- Exception (all regular errors).

3. Catching **specific exceptions** is better than using general Exception.

5. Example Code

(a) Exception Hierarchy in Action

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
except Exception:
    print("Some error occurred")
```

Output:

```
Cannot divide by zero!
```

(b) Catching General Exception

```
try:
    x = int("hello")  # invalid conversion
except Exception as e:
    print("Error:", e)
```

Output:

```
Error: invalid literal for int() with base 10: 'hello'
```

(c) Checking Inheritance

```
print(issubclass(ZeroDivisionError, Exception))  # True
print(issubclass(Exception, BaseException))      # True
```

Output:

```
True
True
```

6. Diagram / Flow

📌 **Exception Hierarchy (Simplified)**

```
BaseException
 ├── SystemExit
 ├── KeyboardInterrupt
```

```
└── Exception
      ├── ArithmeticError
      │     ├── ZeroDivisionError
      │     └── OverflowError
      ├── ValueError
      ├── TypeError
      ├── FileNotFoundError
      └── ...
```

7. Output

- Already shown with examples.
- Exceptions provide detailed error messages for debugging.

8. Common Errors & Debugging

❌ **Error 1: Catching only general Exception**

```
try:
  ...
except Exception:
  print("Error")  # hides real cause
```

✅ Fix: Catch specific exceptions first.

❌ **Error 2: Misunderstanding BaseException**

- Don't use except BaseException: — it also catches KeyboardInterrupt (Ctrl+C).

❌ **Error 3: Ignoring exception messages**

- Always print/log the exception object for debugging.

9. Interview / Industry Insight

- Interview Qs:
  - What is the root of all exceptions in Python? (BaseException).
  - Difference between Exception and BaseException.
  - Example of common exceptions (ValueError, TypeError).
- Industry:

- Exception hierarchy helps in **robust error handling**.
- Example: catching FileNotFoundError specifically in file-handling apps.
- Logging frameworks often use Exception subclasses for better debugging.

5.2: try, except, else, finally

1. Definition / Concept

Python provides structured blocks for exception handling:

- **try** → Defines a block of code where errors may occur.
- **except** → Handles the error if it occurs.
- **else** → Runs if no exceptions occur in the try block.
- **finally** → Always executes (whether error occurs or not), often used for cleanup.

📌 Helps in **graceful error handling** and ensures **important code always runs**.

2. Analogy / Real-Life Connection

Imagine you're **withdrawing money from an ATM**:

- try → Insert card and enter PIN (risky, might fail).
- except → If PIN is wrong, show "Invalid PIN."
- else → If successful, dispense cash.
- finally → Return the card no matter what happens.

3. Syntax

```
try:
    # risky code
except ExceptionType:
    # handle error
else:
    # runs if no error
finally:
    # runs always
```

4. Step-by-Step Explanation

1. **try** → Wrap risky operations.

2. **except** → Catch errors. Can have multiple except blocks.
3. **else** → Only executes when try succeeds.
4. **finally** → Executes regardless of success/failure → often used for cleanup (closing files, releasing resources).

5. Example Code

(a) Simple try-except

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("You cannot divide by zero!")
```

Output:

```
You cannot divide by zero!
```

(b) Multiple except Blocks

```
try:
    x = int("abc")
except ValueError:
    print("Conversion failed: invalid number")
except ZeroDivisionError:
    print("Division error")
```

Output:

```
Conversion failed: invalid number
```

(c) Using else

```
try:
    num = int("123")
except ValueError:
    print("Invalid input")
else:
    print("Conversion successful:", num)
```

Output:

```
Conversion successful: 123
```

(d) Using finally

```
try:
    f = open("test.txt", "w")
    f.write("Hello!")
except Exception as e:
    print("Error:", e)
finally:
    f.close()
    print("File closed.")
```

Output:

File closed.

(e) Full Example (all blocks together)

```
try:
    num = int("42")
    print("Number:", num)
except ValueError:
    print("Invalid number")
else:
    print("Else: No exception occurred")
finally:
    print("Finally: Always executes")
```

Output:

Number: 42

Else: No exception occurred

Finally: Always executes

6. Diagram / Flow

📌 **Control Flow**

```
try → exception?
    ├── Yes → except → finally
    └── No  → else   → finally
```

7. Output

- Shown above.

- Important: finally **always runs**.

8. Common Errors & Debugging

❌ **Error 1: Missing except block**

```
try:
   x = 10 / 0
finally:
   print("Done")
```

Output:

```
Done
ZeroDivisionError: division by zero
```

✅ Fix: Add an except.

❌ **Error 2: Broad except**

```
try:
   ...
except:
   print("Error")  # catches everything, even KeyboardInterrupt
```

✅ Fix: Catch specific exceptions.

❌ **Error 3: Forgetting cleanup without finally**

- Risk: open file/database remains open.

✅ Fix: Use finally or context managers (with).

9. Interview / Industry Insight

- Interview Qs:
  - Difference between else and finally in exception handling.
  - What happens if an error occurs inside finally? (Answer: it overrides the original exception).
- Industry:
  - finally is widely used to **close connections, release locks, cleanup resources**.
  - Example: Always closing a file, even if writing fails.

5.3: Raising Exceptions (raise)

1. Definition / Concept

- **Raising an exception** means **deliberately creating an error condition** in your program using the raise keyword.
- Useful when:
  - You want to **enforce rules**.
  - Signal that **something went wrong**.
  - Create **custom error messages**.

2. Analogy / Real-Life Connection

Imagine a **teacher grading exams**:

- If a student submits an empty answer sheet, the teacher **raises a red flag** saying: *"This is invalid!"*
- Similarly, in Python, raise is used to **signal problems** in logic or input.

3. Syntax

raise ExceptionType("Error message")

Optionally:

raise # re-raises the last exception

4. Step-by-Step Explanation

1. Use raise to throw exceptions explicitly.
2. Can use built-in exceptions (ValueError, TypeError, etc.).
3. Can define **custom exceptions** (subclass of Exception).
4. raise without arguments → re-raises the last caught exception.

5. Example Code

(a) Raising Built-in Exception

```
age = -5
if age < 0:
    raise ValueError("Age cannot be negative")
```

Output:

ValueError: Age cannot be negative

(b) Raising TypeError

```
def divide(a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise TypeError("Both arguments must be numbers")
    return a / b

print(divide(10, 2))
print(divide("10", 2))  # will raise
```

Output:

```
5.0
TypeError: Both arguments must be numbers
```

(c) Re-raising Exception

```
try:
    x = 10 / 0
except ZeroDivisionError as e:
    print("Handling error:", e)
    raise   # re-raises the same exception
```

Output:

```
Handling error: division by zero
ZeroDivisionError: division by zero
```

(d) Custom Exception

```
class InsufficientFundsError(Exception):
    pass

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFundsError("Not enough balance!")
    return balance - amount

print(withdraw(1000, 200))
print(withdraw(1000, 1500))
```

Output:

800
InsufficientFundsError: Not enough balance!

6. Diagram / Flow

📌 **Raise Workflow**

Condition check → If invalid → raise Exception
|
v
Program stops unless handled by try-except

📌 Example:

if age < 0 → raise ValueError

7. Output

- Without handling → program crashes with traceback.
- With try-except → handled gracefully.

8. Common Errors & Debugging

❌ **Error 1: Raising non-exception object**

raise "error" # ❌ not allowed

✅ Fix: Must raise subclass of BaseException.

❌ **Error 2: Forgetting message**

raise ValueError # ❌ message missing

✅ Fix: raise ValueError("Meaningful message")

❌ **Error 3: Using raise outside try-except incorrectly**

- If not handled, program crashes.

✅ Fix: Wrap in try-except if recovery is possible.

9. Interview / Industry Insight

- Interview Qs:
  - How do you raise exceptions in Python?
  - Difference between raising and handling exceptions.
  - Why use custom exceptions?
- Industry:
  - Raising exceptions is common in **APIs, validation logic, financial apps**.
  - Example: Flask/Django raise Http404 or PermissionDenied.
  - Custom exceptions improve **clarity and maintainability**.

5.4: Custom Exception Classes

1. Definition / Concept

- A **custom exception** is a user-defined error class that extends Python's built-in Exception class.
- Used when built-in exceptions (ValueError, TypeError, etc.) are **not descriptive enough** for your application.
- Improves **readability** and **error handling**.

📌 Rule: Custom exceptions should inherit from Exception (or a subclass of it).

2. Analogy / Real-Life Connection

Imagine an **ATM system**:

- Built-in errors (ValueError) → "Invalid amount."
- Custom error (InsufficientFundsError) → "Balance is not enough."
- Custom exceptions give **clearer context** for the application.

3. Syntax

```
class CustomError(Exception):
    def __init__(self, message):
        super().__init__(message)
```

4. Step-by-Step Explanation

1. Create a new class that **inherits from Exception**.
2. Optionally, override __init__ or __str__ to customize error messages.
3. Use raise CustomError("message") to throw it.

4. Handle it with except CustomError: block.

5. Example Code

(a) Simple Custom Exception

```
class NegativeAgeError(Exception):
    pass

def set_age(age):
    if age < 0:
        raise NegativeAgeError("Age cannot be negative")
    print("Age set to:", age)

set_age(25)
set_age(-5)
```

Output:

```
Age set to: 25
NegativeAgeError: Age cannot be negative
```

(b) Custom Exception with Constructor

```
class InsufficientFundsError(Exception):
    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        super().__init__(f"Cannot withdraw {amount}. Balance is {balance}")

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFundsError(balance, amount)
    return balance - amount

print(withdraw(1000, 500))
print(withdraw(1000, 1500))
```

Output:

```
500
InsufficientFundsError: Cannot withdraw 1500. Balance is 1000
```

(c) Handling Custom Exception

```
try:
    raise NegativeAgeError("Age cannot be -10")
except NegativeAgeError as e:
    print("Caught custom exception:", e)
```

Output:

```
Caught custom exception: Age cannot be -10
```

(d) Hierarchy of Custom Exceptions

```
class ApplicationError(Exception):   # Base app error
    pass

class DatabaseError(ApplicationError):
    pass

class ValidationError(ApplicationError):
    pass

try:
    raise ValidationError("Invalid input data")
except ApplicationError as e:
    print("Application error:", e)
```

Output:

```
Application error: Invalid input data
```

6. Diagram / Flow

📌 **Custom Exception Hierarchy**

```
Exception (built-in)
    |
    └── ApplicationError (base for app)
          ├── DatabaseError
          └── ValidationError
```

7. Output

- Custom errors provide **clearer context** in logs and debugging.
- Applications often define a **hierarchy of exceptions**.

8. Common Errors & Debugging

❌ **Error 1: Forgetting to inherit from Exception**

class MyError: # ❌ Wrong

  pass

✅ Fix: class MyError(Exception):

❌ **Error 2: Missing error message**

raise NegativeAgeError # ❌ Less informative

✅ Fix: raise NegativeAgeError("Age cannot be negative")

❌ **Error 3: Too many specific exceptions**

- If you create 50 custom exceptions, code becomes messy.
- ✅ Fix: Use a **base custom exception** and extend only where needed.

9. Interview / Industry Insight

- Interview Qs:
  - How do you create custom exceptions in Python?
  - Why not just use built-in exceptions?
  - Show an example of hierarchy in custom exceptions.
- Industry:
  - Custom exceptions are heavily used in **banking, APIs, ML apps, frameworks**.
  - Example: Django raises ValidationError, PermissionDenied.
  - Custom errors help developers **understand exactly what went wrong**.

5.5: File Handling Basics (opening, reading, writing, appending)

1. Definition / Concept

- **File handling** allows Python programs to **store, retrieve, and manipulate data** in files.
- Modes of opening files:
  - 'r' → Read (default).
  - 'w' → Write (creates new file or overwrites).
  - 'a' → Append (adds to file).
  - 'b' → Binary mode.
  - '+' → Read & write.

📌 **Workflow**: Open → Work (read/write) → Close.

2. Analogy / Real-Life Connection

Think of a **file** like a **notebook**:

- 'r' → Reading notes.
- 'w' → Erasing old content & writing new notes.
- 'a' → Adding new notes at the end.
- Closing the file → Putting the notebook back on the shelf.

3. Syntax

```
file = open("filename.txt", mode)
# operations...
file.close()
```

4. Step-by-Step Explanation

1. Use open("file", "mode") to access a file.
2. Perform operation → read(), write(), append().
3. Always close with file.close() (or use with for auto-close).

5. Example Code

(a) Writing to a File (w)

```
f = open("test.txt", "w")
f.write("Hello, world!\n")
f.write("This is Python file handling.")
f.close()
```

File Content (test.txt):

```
Hello, world!
```

This is Python file handling.

(b) Reading from a File (r)

```
f = open("test.txt", "r")
content = f.read()
print(content)
f.close()
```

Output:

```
Hello, world!
This is Python file handling.
```

(c) Appending to a File (a)

```
f = open("test.txt", "a")
f.write("\nAdding new line at the end.")
f.close()
```

File Content After Appending:

```
Hello, world!
This is Python file handling.
Adding new line at the end.
```

(d) Reading Line by Line

```
f = open("test.txt", "r")
for line in f:
    print(line.strip())
f.close()
```

Output:

```
Hello, world!
This is Python file handling.
Adding new line at the end.
```

(e) Using with (Best Practice)

```
with open("test.txt", "r") as f:
    data = f.read()
    print(data)
# auto closed
```

6. Diagram / Flow

📌 **File Handling Process**

open() → operation (read/write/append) → close()

📌 **Modes**

'r' → read

'w' → write (overwrite)

'a' → append

'r+' → read/write

'b' → binary

7. Output

- Depends on operation.
- Writing/Appending → updates file content.
- Reading → fetches file content.

8. Common Errors & Debugging

❌ **Error 1: File not found**

f = open("nofile.txt", "r") # ❌

Error: FileNotFoundError

✅ Fix: Check path or create file before reading.

❌ **Error 2: Forgetting to close file**

- Causes resource leaks.

  ✅ Fix: Use with open(...) as f:

❌ **Error 3: Wrong mode**

f = open("test.txt", "r")

f.write("hi") # ❌ IOError

✅ Fix: Use correct mode (w or a).

9. Interview / Industry Insight

- Interview Qs:
  - Difference between w and a modes?
  - Why use with open() instead of open()?
  - How to read line by line?
- Industry:
  - File handling is used in **logging, config management, data pipelines**.
  - Best practice → always use with open() for safety.
  - Large-scale apps prefer **databases**, but files still used for **logs & configs**.

5.6: Working with Text and Binary Files

1. Definition / Concept

- **Text files**
  - Store data in **human-readable format** (characters, strings).
  - Example: .txt, .csv.
  - Python reads/writes them as str.
- **Binary files**
  - Store data in **raw bytes** (not human-readable).
  - Example: images (.jpg), audio (.mp3), executables (.exe).
  - Python reads/writes them as bytes.

📌 **Modes**:

- Text mode (default): 'r', 'w', 'a'.
- Binary mode: 'rb', 'wb', 'ab'.

2. Analogy / Real-Life Connection

- **Text file** → Like a **book** written in English → you can read it.
- **Binary file** → Like a **photo** → you can see it with an app, but raw data looks like gibberish.

3. Syntax

```
# Text file
with open("file.txt", "r") as f:
    data = f.read()
```

```
# Binary file
with open("file.jpg", "rb") as f:
    data = f.read()
```

4. Step-by-Step Explanation

1. Use 'r', 'w', 'a' for text mode.
2. Use 'rb', 'wb', 'ab' for binary mode.
3. Text mode deals with **str**, binary mode deals with **bytes**.
4. Always use binary mode for **images, videos, executables**.

5. Example Code

(a) Working with Text File

```
# Writing text
with open("notes.txt", "w") as f:
    f.write("Hello Python!\n")
    f.write("Working with text files.")

# Reading text
with open("notes.txt", "r") as f:
    print(f.read())
```

Output:

```
Hello Python!
Working with text files.
```

(b) Working with Binary File (Copying an Image)

```
# Reading binary file
with open("source.jpg", "rb") as f:
    img_data = f.read()

# Writing binary file
with open("copy.jpg", "wb") as f:
    f.write(img_data)

print("Image copied successfully")
```

Output:

Image copied successfully

(c) Mixing Text and Binary Data (Error Example)

```
with open("notes.txt", "wb") as f:
    f.write("Hello")   # ❌ TypeError: must be bytes
```

✅ Fix: Convert string to bytes

```
f.write("Hello".encode())
```

(d) Reading Binary Data in Chunks

```
with open("source.jpg", "rb") as f:
    chunk = f.read(1024)   # read 1KB
    while chunk:
        print("Read chunk of size:", len(chunk))
        chunk = f.read(1024)
```

6. Diagram / Flow

📌 **Text Mode vs Binary Mode**

Text File: "Hello"
Stored as → ASCII/UTF-8 characters
Read as → str

Binary File: Image (JPG)
Stored as → Raw bytes (0xFFD8FFE0...)
Read as → bytes

7. Output

- Text → readable output.
- Binary → unreadable raw bytes, but essential for non-text data.

8. Common Errors & Debugging

❌ **Error 1: Writing str in binary mode**

- Must use .encode().

❌ **Error 2: Reading binary file in text mode**

- Causes UnicodeDecodeError.

❌ **Error 3: Large binary files**

- Reading entire file may consume memory.
- ✅ Fix: Read/write in chunks.

9. Interview / Industry Insight

- Interview Qs:
  - Difference between text and binary files.
  - When do you use 'rb' vs 'r'?
  - How to copy an image file in Python?
- Industry:
  - **Text files**: configs, logs, CSV.
  - **Binary files**: images, videos, machine learning models.
  - Used in **data science, multimedia apps, system-level programming**.

5.7: Context Managers (with statement)

1. Definition / Concept

- A **context manager** in Python manages resources like files, database connections, and network sockets.
- The most common example → with open(...) as f: for file handling.
- Ensures that resources are **automatically cleaned up**, even if an error occurs.
- Context managers use two special methods:
  - __enter__() → executed when entering the context (with).
  - __exit__() → executed when exiting (cleanup).

2. Analogy / Real-Life Connection

Imagine renting a **hotel room**:

- **Check-in** (__enter__) → You get the room and key.
- **Stay** (your code runs).

- **Check-out** (__exit__) → The hotel automatically cleans up the room, even if you broke something.

3. Syntax

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

Custom context manager:

```
class MyManager:
    def __enter__(self):
        # setup
        return self
    def __exit__(self, exc_type, exc_value, traceback):
        # cleanup
        pass
```

4. Step-by-Step Explanation

1. The with statement calls __enter__() at the start.
2. The returned object (e.g., file handle) is assigned to the variable after as.
3. After the block finishes (success or error), __exit__() is called automatically.
4. Prevents leaks (like unclosed files).

5. Example Code

(a) File Handling with with

```
with open("notes.txt", "w") as f:
    f.write("Hello using context manager!")

print("File is automatically closed:", f.closed)
```

Output:

File is automatically closed: True

(b) Reading File with with

```
with open("notes.txt", "r") as f:
    print(f.read())
```

Output:

Hello using context manager!

(c) Custom Context Manager Class

```
class MyContext:
    def __enter__(self):
        print("Entering context")
        return self
    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting context")
        if exc_type:
            print("Error occurred:", exc_value)
        return True  # suppress exception

with MyContext():
    print("Inside block")
    # raise ValueError("Test error")  # try uncommenting
```

Output:

Entering context

Inside block

Exiting context

(If error raised → it's handled by __exit__).

(d) Context Manager with contextlib

```
from contextlib import contextmanager

@contextmanager
def my_manager():
    print("Setup")
    yield
    print("Cleanup")

with my_manager():
    print("Doing work inside context")
```

Output:

Setup

Doing work inside context
Cleanup

6. Diagram / Flow

📌 **With Flow**

```
with open("file.txt") as f:
   |
   ├── f.__enter__()  → open file
   ├── block executes
   └── f.__exit__()   → close file
```

7. Output

- Files closed automatically.
- Custom managers allow controlled resource handling.

8. Common Errors & Debugging

❌ **Error 1: Forgetting to close file without with**

```
f = open("file.txt")
# forgot f.close() ❌
```

✅ Fix: Use with.

❌ **Error 2: Misunderstanding __exit__ return**

- If __exit__ returns True, exception is suppressed.
- If False/None, exception propagates.

❌ **Error 3: Using context manager incorrectly**

```
with "text" as f:  # ❌ not an object with __enter__ and __exit__
```

✅ Fix: Only objects implementing context manager protocol.

9. Interview / Industry Insight

- Interview Qs:
  - What is a context manager?
  - Difference between with and try-finally?
  - How do you write a custom context manager?
- Industry:
  - Context managers used in **files, DB connections, threading locks, APIs**.
  - Examples:
    - with open(...) for files.
    - with sqlite3.connect(...) for databases.
    - with threading.Lock(): for concurrency.

5.8: os and pathlib Modules for File Paths

1. Definition / Concept

- **os module** → Provides functions to interact with the **operating system** (file paths, directories, environment variables).
- **pathlib module** → Object-oriented approach to handling **file system paths**.
- Both are used for **file navigation, creation, deletion, and path manipulations**.

📌 os is older & procedural.

📌 pathlib is modern, introduced in Python 3.4 (preferred).

2. Analogy / Real-Life Connection

Think of your **computer's file explorer**:

- **os** → Like typing commands in terminal (cd, mkdir, ls).
- **pathlib** → Like using a GUI explorer with objects (folders/files) that you can easily manipulate.

3. Syntax

```
import os
from pathlib import Path
```

4. Step-by-Step Explanation

1. Use **os** for functions like os.getcwd(), os.listdir(), os.remove().

2. Use **pathlib.Path** objects for path manipulation.
3. Both can create, delete, join, and check files & directories.

5. Example Code

(a) Using os Module

```
import os

print("Current directory:", os.getcwd())

# List files
print("Files:", os.listdir())

# Make directory
os.mkdir("test_folder")

# Rename file/folder
os.rename("test_folder", "renamed_folder")

# Remove folder
os.rmdir("renamed_folder")
```

(b) File Path with os.path

```
file_path = os.path.join("folder", "file.txt")
print("Joined path:", file_path)

print("Absolute path:", os.path.abspath("file.txt"))
print("Exists:", os.path.exists("file.txt"))
print("Is file:", os.path.isfile("file.txt"))
print("Is dir:", os.path.isdir("folder"))
```

(c) Using pathlib (Modern Approach)

```
from pathlib import Path

p = Path("example.txt")
```

```
# Create file
p.write_text("Hello from pathlib!")

# Read file
print(p.read_text())

# Check existence
print("Exists?", p.exists())

# File details
print("File name:", p.name)
print("Parent folder:", p.parent)

# Create directory
Path("new_dir").mkdir(exist_ok=True)
```

(d) Iterating Files in Directory (pathlib)

```
path = Path(".")
for file in path.iterdir():
    print(file)
```

(e) File Extension Filtering

```
for file in Path(".").glob("*.txt"):
    print("Text file:", file)
```

6. Diagram / Flow

📌 **os vs pathlib**

```
os.path.join("folder", "file.txt")    → procedural
Path("folder") / "file.txt"           → object-oriented
```

📌 Example:

```
os → "C:/Users/User/file.txt"
pathlib → Path("C:/Users/User/file.txt")
```

7. Output

- os → Returns **strings** for paths.
- pathlib → Returns **Path objects**, easier to manipulate.

8. Common Errors & Debugging

❌ **Error 1: Using relative paths incorrectly**

open("file.txt") # ❌ may fail if not in current dir

✅ Fix: Use absolute paths with os.path.abspath() or Path.resolve().

❌ **Error 2: Removing non-empty directory with os.rmdir()**

- Fails if directory has files.

  ✅ Fix: Use shutil.rmtree() or Path("dir").rmdir() only when empty.

❌ **Error 3: Mixing os and pathlib improperly**

- Stick to one style consistently.

9. Interview / Industry Insight

- Interview Qs:
  - Difference between os and pathlib?
  - How do you check if a path exists?
  - How do you list all .txt files in a directory?
- Industry:
  - **os** still used in legacy projects.
  - **pathlib** is preferred for new projects (cleaner, OOP-based).
  - File path manipulation is crucial in **data science, ML pipelines, automation scripts, web apps**.

5.9: JSON File Handling (json module`)

1. Definition / Concept

- **JSON (JavaScript Object Notation)** → lightweight data format for **storing & exchanging structured data**.
- Python provides the **json module** to work with JSON.

- JSON ↔ Python mappings:

JSON	Python
Object	dict
Array	list
String	str
Number	int/float
true/ false	True/ False
null	None

2. Analogy / Real-Life Connection

Think of JSON as a **menu card** in a restaurant:

- Structured and easy to read.
- Can be exchanged between **customer (frontend)** and **chef (backend)**.
- Python uses json to **convert menu card (JSON) into Python dictionary** and back.

3. Syntax

```
import json

# Python → JSON
json.dumps(python_object)

# JSON → Python
json.loads(json_string)

# File handling
json.dump(python_object, file)
json.load(file)
```

4. Step-by-Step Explanation

1. Use **json.dumps()** → convert Python object to JSON string.
2. Use **json.loads()** → convert JSON string to Python object.
3. Use **json.dump()** → write JSON to file.
4. Use **json.load()** → read JSON from file.

5. Example Code

(a) Python to JSON String

```
import json

student = {"name": "Alice", "age": 21, "marks": [85, 90, 95]}
json_str = json.dumps(student)
print(json_str)
```

Output:

```
{"name": "Alice", "age": 21, "marks": [85, 90, 95]}
```

(b) JSON String to Python Object

```
json_data = '{"name": "Bob", "age": 22, "is_student": true}'
python_obj = json.loads(json_data)
print(python_obj)
print(python_obj["name"])
```

Output:

```
{'name': 'Bob', 'age': 22, 'is_student': True}
Bob
```

(c) Writing JSON to File

```
student = {"name": "Charlie", "age": 23, "course": "Python"}

with open("student.json", "w") as f:
    json.dump(student, f)
```

File student.json:

```
{"name": "Charlie", "age": 23, "course": "Python"}
```

(d) Reading JSON from File

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

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

Output:

```
{'name': 'Charlie', 'age': 23, 'course': 'Python'}
```

Python

(e) Pretty Printing JSON

```
student = {"name": "David", "age": 24, "marks": [70, 80, 90]}
print(json.dumps(student, indent=4))
```

Output:

```
{
   "name": "David",
   "age": 24,
   "marks": [
      70,
      80,
      90
   ]
}
```

6. Diagram / Flow

📌 **JSON ↔ Python Conversion**

Python dict → json.dumps() → JSON string
Python dict → json.dump() → JSON file

JSON string → json.loads() → Python dict
JSON file → json.load() → Python dict

7. Output

- JSON strings look like JavaScript objects.
- Python converts them into dictionaries and lists automatically.

8. Common Errors & Debugging

❌ **Error 1: Using single quotes in JSON**

```
json.loads("{'name':'Alice'}")  # ❌ invalid JSON
```

✅ Fix: JSON requires **double quotes**.

❌ **Error 2: Writing non-serializable objects**

```
import datetime

json.dumps(datetime.datetime.now())  # ❌ not supported
```

✅ Fix: Convert to string → str(datetime.datetime.now()).

❌ **Error 3: Forgetting to close file**

- Always use with open(...) to avoid leaks.

9. Interview / Industry Insight

- Interview Qs:
  - Difference between json.dumps() and json.dump().
  - How to parse JSON into Python dict?
  - How does JSON handle null?
- Industry:
  - JSON is the **standard format for APIs** (REST, GraphQL).
  - Widely used in **web apps, ML models (config files), log storage**.
  - Almost every modern app exchanges data in JSON.

5.10: Reading/Writing CSV & Excel using Pandas

1. Definition / Concept

- **CSV (Comma-Separated Values)** → Text file format where data is stored in rows & columns separated by commas.
- **Excel (.xlsx)** → Spreadsheet format widely used in business and data analysis.
- **Pandas** → Python library that provides easy functions to **read, write, and process** CSV/Excel files.

📌 Functions:

- pd.read_csv() → Load CSV.
- DataFrame.to_csv() → Save to CSV.
- pd.read_excel() → Load Excel.
- DataFrame.to_excel() → Save to Excel.

2. Analogy / Real-Life Connection

Think of **CSV/Excel files** like **class registers**:

- Each row = student record.
- Each column = attribute (name, roll no, marks).
- Pandas is like an **assistant** that can instantly load, modify, and save registers.

3. Syntax

```
import pandas as pd

# Reading CSV
df = pd.read_csv("file.csv")

# Writing CSV
df.to_csv("output.csv", index=False)

# Reading Excel
df = pd.read_excel("file.xlsx")

# Writing Excel
df.to_excel("output.xlsx", index=False)
```

4. Step-by-Step Explanation

1. Import **pandas** (import pandas as pd).
2. Use pd.read_csv() or pd.read_excel() to load file into a **DataFrame**.
3. Work with DataFrame (filter, analyze, transform).
4. Use to_csv() or to_excel() to save results.

5. Example Code

(a) Reading a CSV File

```
import pandas as pd

df = pd.read_csv("students.csv")
print(df.head())   # show first 5 rows
```

Example students.csv:

```
name,age,marks
```

Alice,20,85
Bob,22,90
Charlie,21,88

Output:

```
     name  age  marks
0   Alice   20     85
1     Bob   22     90
2 Charlie   21     88
```

(b) Writing DataFrame to CSV

```
data = {"name": ["David", "Eva"], "age": [23, 21], "marks": [92, 95]}
df = pd.DataFrame(data)
df.to_csv("new_students.csv", index=False)
```

Generated File (new_students.csv):

name,age,marks
David,23,92
Eva,21,95

(c) Reading an Excel File

```
df = pd.read_excel("students.xlsx")
print(df)
```

Output (example):

```
     name  age  marks
0   Alice   20     85
1     Bob   22     90
2 Charlie   21     88
```

(d) Writing DataFrame to Excel

```
df.to_excel("output.xlsx", index=False)
print("Excel file created")
```

(e) Reading CSV with Options

```
df = pd.read_csv("students.csv", usecols=["name", "marks"])
print(df)
```

Output:

```
    name  marks
0   Alice     85
1     Bob     90
2 Charlie     88
```

6. Diagram / Flow

📌 **File ↔ Pandas ↔ File**

CSV/Excel file → pd.read_*() → DataFrame → df.to_*() → CSV/Excel file

7. Output

- Data is stored in Pandas DataFrames → tabular format.
- Easy to save changes back into CSV/Excel.

8. Common Errors & Debugging

❌ **Error 1: File not found**

pd.read_csv("nofile.csv")

✅ Fix: Ensure correct file path.

❌ **Error 2: Writing with index column unintentionally**

df.to_csv("output.csv") # adds extra index column

✅ Fix: Use index=False.

❌ **Error 3: Missing Excel engine**

pd.read_excel("file.xlsx") # ❌ requires openpyxl

✅ Fix: Install with → pip install openpyxl.

9. Interview / Industry Insight

- Interview Qs:
  - How do you read/write CSV & Excel files using Pandas?
  - What's the difference between read_csv and read_excel?
  - Why do we use index=False in to_csv()?

- Industry:
  - CSV/Excel handling is crucial in **data analysis, machine learning, business reporting**.
  - Pandas is the **standard tool** in data pipelines.
  - Often used for **ETL (Extract, Transform, Load)** processes.

5.11: Handling Data-Related Errors (pd.errors)

1. Definition / Concept

- When using **Pandas** to read CSV/Excel files, sometimes data issues cause errors.
- These errors are raised as exceptions from **pandas.errors** (aliased as pd.errors).
- They help identify **specific problems in data handling** instead of generic errors.

📌 Common Pandas errors:

- pd.errors.EmptyDataError → File is empty.
- pd.errors.ParserError → Malformed CSV (bad delimiter/format).
- pd.errors.DtypeWarning → Mixed data types in a column.

2. Analogy / Real-Life Connection

Imagine **checking exam papers**:

- If the paper is **blank** → EmptyDataError.
- If answers are **scrambled/malformed** → ParserError.
- If handwriting is **inconsistent** (mix of numbers & words in marks column) → DtypeWarning.

3. Syntax

```
import pandas as pd

try:
    df = pd.read_csv("file.csv")
except pd.errors.EmptyDataError:
    print("File is empty")
except pd.errors.ParserError:
    print("File is malformed")
```

4. Step-by-Step Explanation

1. Use try-except to wrap Pandas file operations.
2. Catch **specific pd.errors exceptions** for better error handling.
3. Warnings (like DtypeWarning) can be suppressed or fixed with arguments like dtype.

5. Example Code

(a) Empty File Error

```
import pandas as pd

try:
    df = pd.read_csv("empty.csv")
except pd.errors.EmptyDataError:
    print("Error: The file is empty!")
```

Output:

```
Error: The file is empty!
```

(b) Parser Error (Bad Format)

```
from io import StringIO

data = "name,age\nAlice,20\nBob"  # missing value
try:
    df = pd.read_csv(StringIO(data), error_bad_lines=True)
except pd.errors.ParserError:
    print("Error: Malformed CSV file")
```

Output:

```
Error: Malformed CSV file
```

(c) Dtype Warning (Mixed Types)

```
import warnings
warnings.filterwarnings("ignore")  # suppress warnings

df = pd.read_csv("mixed_types.csv", dtype={"marks": str})
print(df.head())
```

(d) Handling Multiple Errors

```
try:
```

```
    df = pd.read_csv("students.csv")
except pd.errors.EmptyDataError:
    print("File is empty")
except pd.errors.ParserError:
    print("CSV parsing failed")
except FileNotFoundError:
    print("File not found")
```

6. Diagram / Flow

📌 **Pandas File Handling Errors**

```
read_csv/read_excel
    |
    ├── File empty? → EmptyDataError
    ├── Malformed CSV? → ParserError
    ├── Mixed types? → DtypeWarning
    └── File missing? → FileNotFoundError (from Python)
```

7. Output

- Pandas errors are descriptive → make debugging file issues easier.

8. Common Errors & Debugging

❌ **Error 1: Ignoring warnings**

- DtypeWarning may indicate serious data quality issues.

✅ Fix: Convert column explicitly → dtype={"col": str}.

❌ **Error 2: Confusing Pandas errors with Python errors**

- FileNotFoundError is a **Python error**, not pd.errors.

❌ **Error 3: Malformed CSV**

- Pandas may silently skip lines if not careful.

✅ Fix: Use error_bad_lines=True (older Pandas) or on_bad_lines="error" (newer Pandas).

9. Interview / Industry Insight

- Interview Qs:
  - What is pd.errors.EmptyDataError?
  - How do you handle malformed CSV in Pandas?
  - What causes DtypeWarning and how to fix it?
- Industry:
  - Handling **dirty data** is crucial in real-world projects.
  - Pandas error handling ensures data pipelines don’t crash unexpectedly.
  - In **ETL jobs, ML preprocessing, and big data pipelines**, these checks are mandatory.