

# Week 2 – Exception Handling & File I/O

Add a brief explanation of the presentation's purpose

# Learning Objectives

By the end of this week, students will be able to:

1. Understand the purpose of exception handling and apply try, except, finally, and raise.
2. Implement error logging to record runtime issues.
3. Perform file operations (read, write, append) safely using Python.
4. Develop a mini-application that integrates exception handling with file I/O.

# What Happens if a Program Crashes?

## Possible Consequences of a Crash

### Loss of Data

- Unsaved work disappears.
- Example: Writing an essay in Word and the program shuts down → you lose your progress.

### Corruption of Files

- Files being read or written at the time of the crash may become damaged.
- Example: An image file becomes unreadable if the program quits while saving.

### Interrupted Transactions

- If the program is handling financial or business transactions, these may be incomplete or duplicated.
- Example: ATM debits your account but fails to dispense cash.

### Bad User Experience

- Users lose trust in the software.
- Example: An e-commerce checkout crash may cause customers to abandon purchases.

### System Instability

- A crash in one program can sometimes affect the operating system or other apps.
- Example: A driver crash causes the whole computer to freeze.

# Real-World Examples of Error Handling

Introduce three **concrete industry cases** students can relate to:

- **ATM Transaction Logs**
  - ATMs use **logs** to record each step (card insert, PIN check, debit, cash dispense).
  - If the ATM crashes mid-transaction, the log helps **reverse or fix errors**.
- **Online Form Submissions**
  - Websites save drafts in the background (auto-save, session management).
  - Even if the browser crashes, **data is preserved**.
- **File Corruption Handling**
  - Microsoft Word/Google Docs create **auto-recovery backups**.
  - If the app crashes, the user can reopen the file without losing all progress.

# Exception Handling Basics

## Why Errors Occur

- **Syntax Errors** → mistakes in code structure (detected before running).

```
python

# Missing colon
if True
    print("Hello")
```

- **Runtime Errors** → occur during execution (unexpected input/conditions).

```
python

num = int("abc")  # ValueError at runtime
```

Syntax errors stop code before it runs. Runtime errors happen while running → these are where *exception handling* is crucial.

# try, except, else, finally, raise

When writing programs, errors are **unavoidable**. A user might enter text instead of a number, a file may not exist, or a calculation might be impossible (like dividing by zero). Without safeguards, these errors cause the program to **crash**.

Python gives us special keywords to handle these problems gracefully:

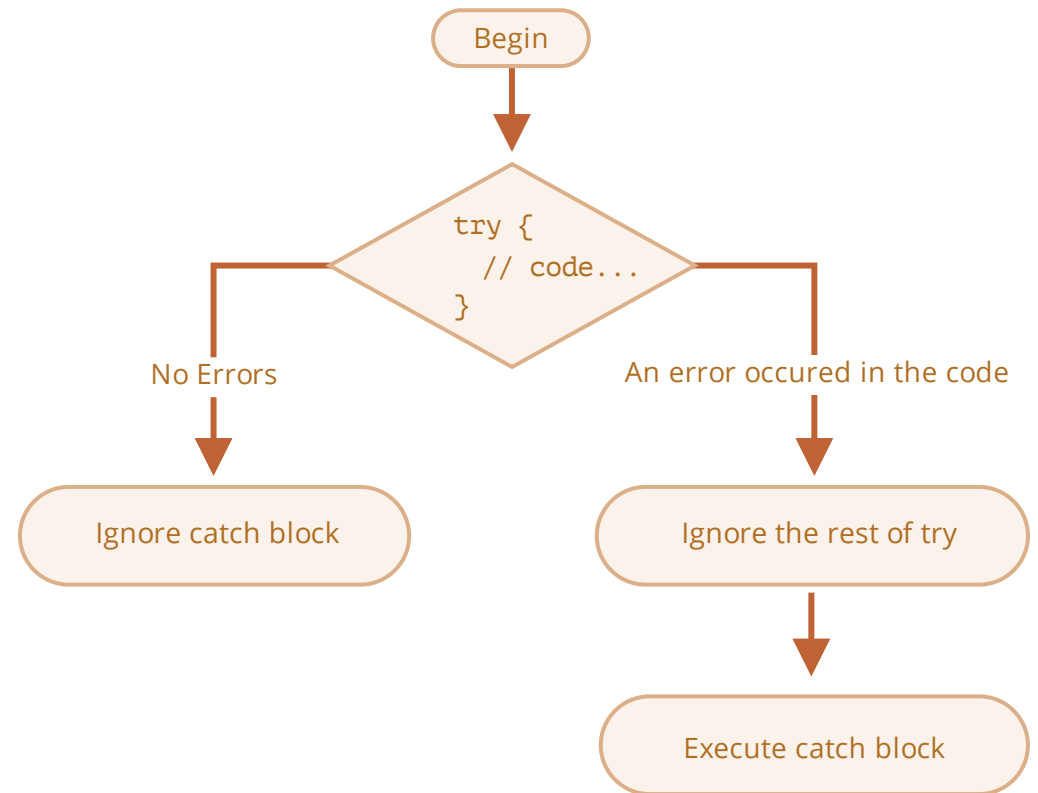
- **try** → Defines a block of code that *might* cause an error.
- **except** → Catches and handles the error so the program doesn't crash.
- **else** → Runs if no errors occur inside the try block.
- **finally** → Runs no matter what happens (error or no error); often used for cleanup.
- **raise** → Allows the programmer to trigger an error on purpose when certain conditions are not met.

# try

- **What it does:** Wraps the code that *might* cause an error.
- If no error → runs normally.
- If error → jumps to the appropriate.

python

```
try:  
    number = int("123")    # safe conversion  
    print(number)
```



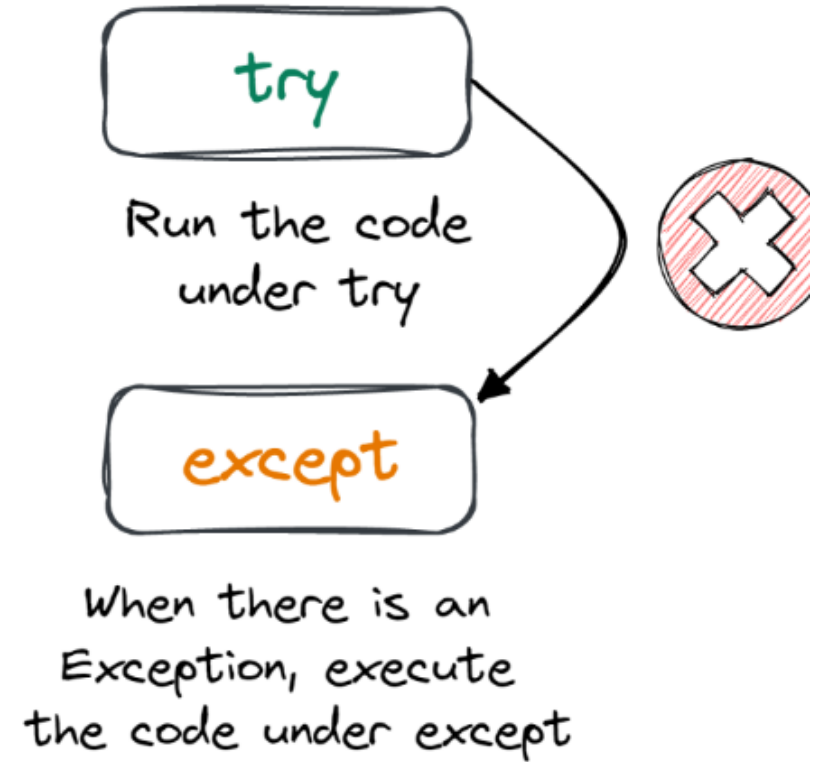
# except

---

**What it does:** Catches and handles the error so the program doesn't crash.

- You can catch specific errors (e.g., `ValueError`) or all errors.

```
try:  
    number = int("abc")    # invalid conversion  
except ValueError:  
    print("That's not a valid number!")
```



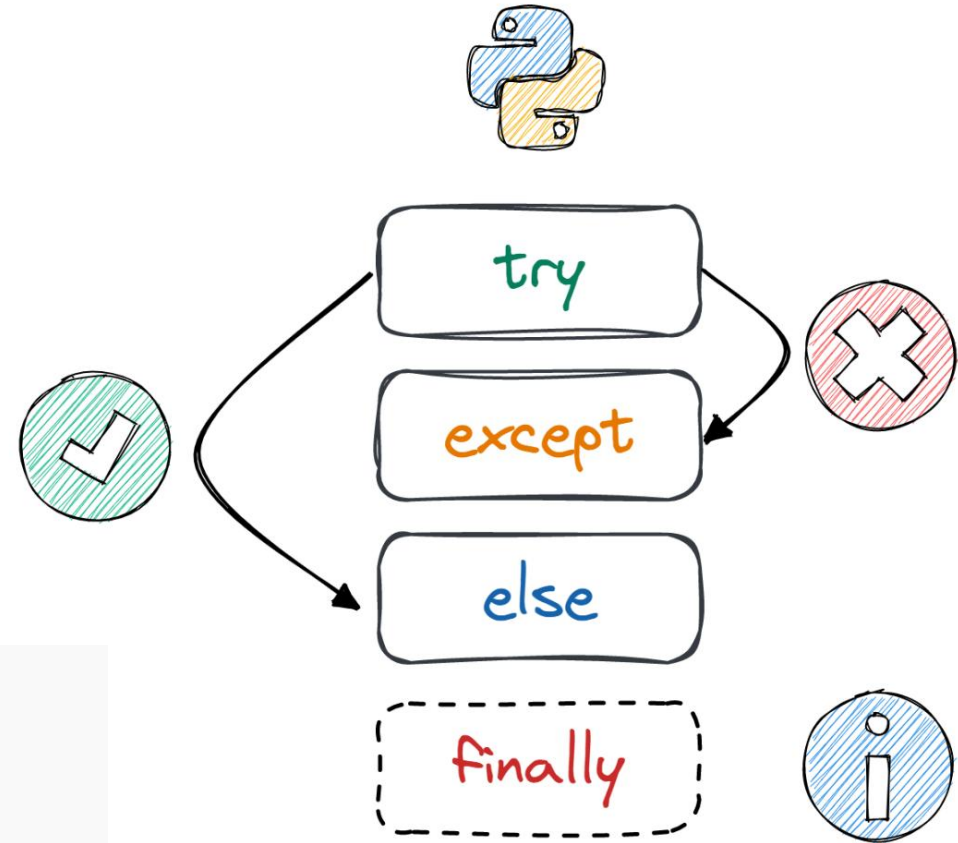


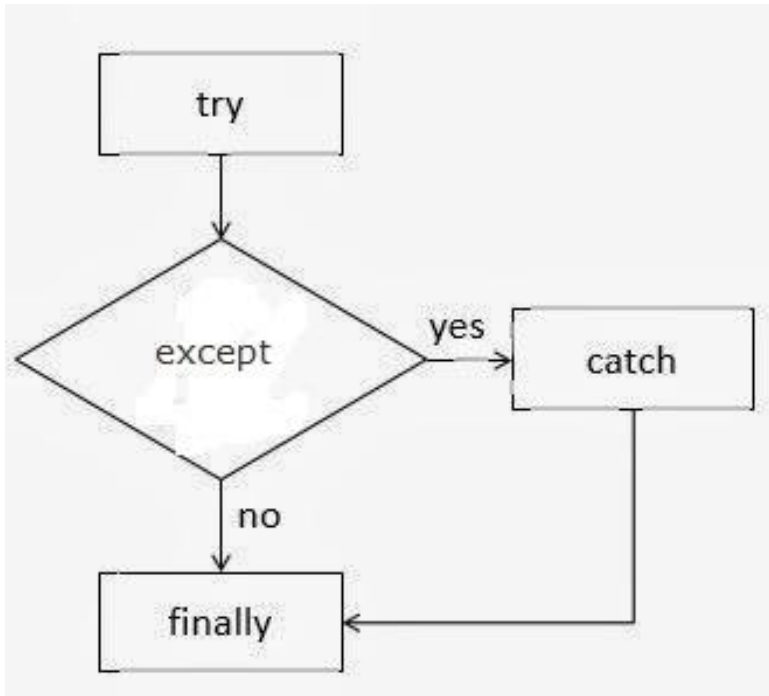
# else

**What it does:** Runs **only** if no **exception** happened in the try block.

- Useful for code that should only run when everything goes smoothly.

```
try:  
    x = 10 / 2  
except ZeroDivisionError:  
    print("You can't divide by zero.")  
else:  
    print("Division successful:", x)
```





# finally

---

**What it does:** Runs **always**, whether there was an error or not.

- Often used for cleanup tasks (closing files, releasing resources).

```
try:
    f = open("data.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("File not found!")
finally:
    print("Execution complete. Closing resources if needed.")
```

```
try:
    num = int(input("Enter a number: "))
    print("The square is:", num * num)
except ValueError:
    print("Invalid input. Please enter a number.")
```

## Exercise 1 – Safe Number Input

### Task:

Write a program that asks the user for a number.

- If the user enters text instead of a number, handle the error with `except`.
- If the number is valid, print its square.

```
try:
    age = int(input("Enter your age: "))
    if age < 0:
        raise ValueError("Age cannot be negative!")
    print("You are", age, "years old.")
except ValueError as e:
    print("Error:", e)
```

## Exercise 3 – Age Validator with Raise

### Task:

Write a program that asks the user for a number.

- If the age is negative, use `raise` to throw a `ValueError`.
- Handle the error with `except` and print a custom message.
- If valid, print "You are X years old."

# Error Logging in Python

In programming, errors are unavoidable — files go missing, users enter invalid data, or systems crash unexpectedly. As beginners, we often rely on `print()` to quickly check what went wrong. But here's the problem:

- `print()` is **temporary** — once the program ends, those messages are gone.
- It gives **no timestamps** — you can't tell *when* the error happened.
- It lacks **structure** — everything looks the same, whether it's a warning or a critical failure.

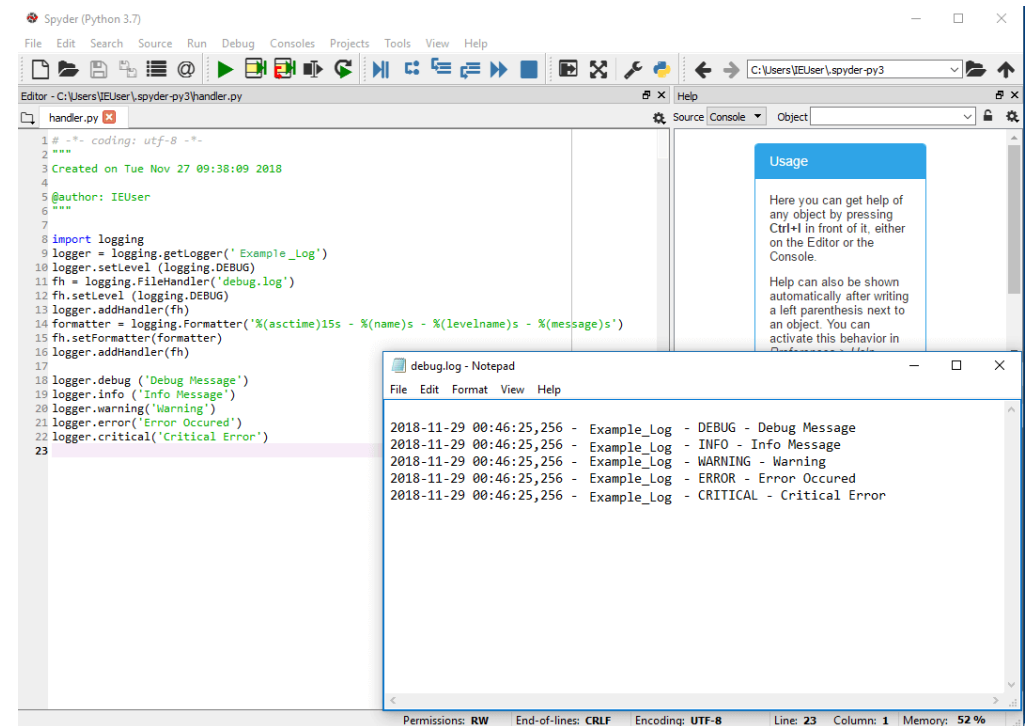
In real-world applications, especially in **banking systems, servers, or cybersecurity tools**, this is not enough. We need something more reliable.

# Error Logging in Python

That's where **logging** comes in.

- Logging allows us to:
- Keep a **permanent record** of errors.
- Add **timestamps** for when issues occur.
- Record **severity levels** (DEBUG, INFO, WARNING, ERROR, CRITICAL).
- Provide an **audit trail** useful for debugging and compliance.

*Print tells you what's happening right now. Logging helps you understand what happened yesterday, last week, or even months ago.*



The screenshot shows the Spyder Python IDE interface. The main editor window displays a Python script named `handler.py` that configures a logging module to write to a file named `debug.log`. The script includes comments, imports, and code to set the logging level to `DEBUG`, add a file handler, and log messages at `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL` levels.

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Tue Nov 27 09:38:09 2018
4
5 @author: IEUser
6 """
7
8 import logging
9 logger = logging.getLogger('Example_Log')
10 logger.setLevel(logging.DEBUG)
11 fh = logging.FileHandler('debug.log')
12 fh.setLevel(logging.DEBUG)
13 logger.addHandler(fh)
14 formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
15 fh.setFormatter(formatter)
16 logger.addHandler(fh)
17
18 logger.debug('Debug Message')
19 logger.info('Info Message')
20 logger.warning('Warning')
21 logger.error('Error Occured')
22 logger.critical('Critical Error')
23
```

A Notepad window titled `debug.log - Notepad` is open in the foreground, displaying the output of the logging script. The output shows five log entries, each with a timestamp, the logger name, the severity level, and the message.

Timestamp	Logger Name	Severity Level	Message
2018-11-29 00:46:25,256	Example_Log	DEBUG	Debug Message
2018-11-29 00:46:25,256	Example_Log	INFO	Info Message
2018-11-29 00:46:25,256	Example_Log	WARNING	Warning
2018-11-29 00:46:25,256	Example_Log	ERROR	Error Occured
2018-11-29 00:46:25,256	Example_Log	CRITICAL	Critical Error

# Why Use Logging?

The **logging module** in Python is designed to keep a **permanent record** of events.

Advantages of Logging:

- **Permanent Record** → Saved in a file for later review.
- **Timestamps** → Tells you when it happened.
- **Severity Levels** → DEBUG, INFO, WARNING, ERROR, CRITICAL.
- **Audit Trail** → Important for compliance, especially in security and financial apps.
- **Flexibility** → Can log to files, servers, or monitoring dashboards.

# Demo: Using Python's Logging Module

---

## Step 1: Import and Configure

```
import logging

logging.basicConfig(
    filename="app.log",           # Log file name
    level=logging.DEBUG,         # Capture all levels
    format="%(asctime)s - %(levelname)s - %(message)s"
)
```



# Demo: Using Python's Logging Module

---

Step 2: Replace print() with logging

```
try:
    num = int("abc") # This will raise ValueError
except ValueError as e:
    print("Error occurred:", e)          # Old way
    logging.error("Error occurred: %s", e) # New way
```

# Demo: Using Python's Logging Module

Step 3: Check app.log

```
2025-09-27 12:15:03,145 - ERROR - Error occurred: invalid literal for int() with base 10: 'abc'
```

# Demo: Using Python's Logging Module

---

## Step 4: Using Different Log Levels

```
logging.debug("This is a debug message")    # Developer details
logging.info("Program started successfully") # General info
logging.warning("Low disk space detected")   # Warning
logging.error("File not found error")        # Recoverable error
logging.critical("System crash!")           # Severe issue
```

# Activity (Hands-On) (20 minutes)

## Task

Write a program that:

- Asks the user for two numbers.
- Divides them.
- Logs all errors (invalid input, division by zero).
- Writes logs into calculator.log.
- Submit to your github account and provide link

# File I/O Operations in Python

In programming, data stored in variables and memory is **temporary** — it disappears when the program ends. To make data **permanent**, programs must store it in files on the computer's storage device. This process of reading data from files (**Input**) and saving data to files (**Output**) is called **File I/O (Input/Output) Operations**.

Python provides a simple yet powerful way to work with files using built-in functions such as `open()`, `read()`, and `write()`. With these operations, we can:

- Save user input or program output to a file for later use.
  - Load data from existing files into our program.
  - Update or append new information to files.
  - Handle different file formats such as text files (.txt) and binary files (images, audio, etc.).
- For example, a text editor uses file I/O to save your notes to a .txt file, and a banking application records each transaction in a log file. Without file I/O, programs would lose all information every time they are closed.

Mode	Meaning	Example
r	Read (default) – file must exist	<code>open("data.txt", "r")</code>
w	Write – creates new file or overwrites existing	<code>open("data.txt", "w")</code>
a	Append – adds to end of file	<code>open("data.txt", "a")</code>
rb	Read binary file	<code>open("image.png", "rb")</code>
wb	Write binary file	<code>open("file.bin", "wb")</code>

## File Modes

In Python, files can be opened using `open(filename, mode)`.

# Opening and Closing Files

Traditional method:

```
file = open("example.txt", "w")  
file.write("Hello, File I/O!\n")  
file.close()
```

Problem: If the program crashes, the file may not close properly → **data corruption.**

# Safer Way: `with` Statement

- The file **closes automatically** when the block ends, even if an error occurs.

```
with open("example.txt", "w") as file:  
    file.write("This is safer!\n")
```



# Reading from Files

`read()` → Reads the entire file

python

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

`readline()` → Reads one line at a time

python

```
with open("example.txt", "r") as file:  
    line = file.readline()  
    print(line)
```

`readlines()` → Reads all lines into a list

python

```
with open("example.txt", "r") as file:  
    lines = file.readlines()  
    for line in lines:  
        print(line.strip())
```

# Writing to Files

## Overwriting (w)

python

```
with open("notes.txt", "w") as file:  
    file.write("First line\n")  
    file.write("Second line\n")
```

## Appending (a)

python

```
with open("notes.txt", "a") as file:  
    file.write("This will be added at the end\n")
```

# Deliverable:

## **Expense Tracker with Error Handling**

### Requirements

#### **1. User Menu**

- The program should present a simple text-based menu:

1. Add Expense
2. View All Expenses
3. Search Expense by Date
4. Exit

# Deliverable: Expense Tracker with Error Handling

## Requirements

### 2. Features

#### •Add Expense

- Ask user for: date (YYYY-MM-DD), category, amount.

#### •Handle invalid input:

- Wrong date format → show error message.
- Non-numeric amount → show error message.
- Negative amounts → not allowed, raise error.

#### •View All Expenses

- Display records from expenses.csv.
- If file missing, create it with headers (Date,Category,Amount).

#### •Search Expense by Date

- Ask for a date and display all matching entries.
- Handle invalid dates or no matching records gracefully.

#### •Exit

- End program safely.

# Error Handling Requirements

- Use `try`, `except`, `else`, and `finally` where appropriate.
- Catch and handle:
  - `ValueError` (invalid numbers, invalid date input).
  - `FileNotFoundError` (if `expenses.csv` doesn't exist).
- Use `raise` to enforce rules:
- Negative amounts → `raise ValueError("Amount cannot be negative.")`.

## Deliverables

**Source Code →**  
`expense_tracker.py`

**Sample Data File →**  
`expenses.csv`

Upload to your github  
account and send link



End-of-Presentation