🐍 Problem Solving Using Python

📖 Unit III – Exception Handling and Libraries

Topics Covered:

- **Introduction to exceptions and errors**
- Handling exceptions using `try-except-else-finally`
- **Introduction to Pandas**: DataFrames and Series
- Basic operations using Pandas
- Reading and writing CSV files using Pandas

🧑‍🏫 Prepared by:

Mr. Mohammed Siraj B
School of Engineering
St Aloysius (Deemed to be University), Mangalore

Academic Year: 2025 – 2026

💡 Notes Features:

✅ Detailed explanations with simple examples
✅ Step-by-step walkthrough of concepts
✅ Practical exercises for hands-on practice
✅ Beautifully structured and easy-to-read format

🌟 **“Learning Python is the first step to solving real-world problems efficiently!”** 🌟

⚙️ Unit III – Exception Handling and Libraries

🧩 1️⃣ Introduction to Errors and Exceptions

💡 What is an Error?

An **error** is a problem in a program that **prevents it from executing properly**.
Errors can occur due to:

- Wrong syntax
- Wrong operations (like dividing by zero)
- Missing files or invalid inputs

When Python finds an error, it **stops the program immediately** and shows an **error message**.

🧠 Types of Errors

Type	Description	Example
Syntax Error	Occurs when the code violates Python syntax rules	`print("Hello"`
Runtime Error (Exception)	Occurs while executing a correct program	`10 / 0`
Logical Error	Code runs, but gives incorrect result	Using `+` instead of `-`

🧩 Example: Syntax Error

```
# Missing closing parenthesis
print("Welcome to Python"
```

🖥️ **Output:**

```
SyntaxError: unexpected EOF while parsing
```

🧩 **Explanation:**
Python detects the syntax mistake **before execution** and stops the program.

🧩 Example: Runtime Error

```
a = 10
b = 0
print(a / b)
```

🖥️ **Output:**

```
ZeroDivisionError: division by zero
```

🧩 **Explanation:**
The program runs but crashes at runtime when division by zero occurs.

💥 What Happens When an Error Occurs?

1. Python stops normal execution.
2. Creates an **exception object** describing the error.
3. If not handled, it **terminates the program** with an error message.

⚠️ 2️⃣ What is an Exception?

💡 Definition

An **Exception** is an **event** that occurs during program execution that disrupts the normal flow.

Example:

- Dividing by zero
- Accessing a file that doesn't exist
- Converting text to integer (invalid conversion)

🧱 Difference Between Error and Exception

Feature	Error	Exception

Feature	Error	Exception
Definition	Problems in the code itself	Issues during runtime
When occurs	Before execution	During execution
Can be handled?	❌ No	✅ Yes
Example	`SyntaxError`	`ZeroDivisionError` , `ValueError`

🎯 3️⃣ Need for Exception Handling

When exceptions are not handled, the program **crashes** suddenly.
To prevent this, we use **exception handling** to:

✅ Handle errors gracefully
✅ Display friendly messages
✅ Continue program execution safely

🧱 4️⃣ Handling Exceptions using `try` and `except`

💡 Concept Explanation

The `try` and `except` blocks are used to handle exceptions in Python.

- Code that might cause an error → written inside `try` block
- Code that handles the error → written inside `except` block

⚙️ Syntax

```
try:
    # Code that may cause an error
except:
    # Code to handle the error
```

🧩 Example 1: Simple Try-Except

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    print("Result:", a / b)
except:
    print("⚠️ Oops! Something went wrong.")
```

🖥️ Output (if user enters b = 0):

```
⚠️ Oops! Something went wrong.
```

🧩 Explanation:

The program doesn't crash even if division by zero occurs.
The error is handled gracefully by the `except` block.

🧩 Example 2: Handling Specific Exceptions

```
try:
    num = int(input("Enter number: "))
    result = 10 / num
    print("Result:", result)
except ZeroDivisionError:
    print("❌ You cannot divide by zero!")
except ValueError:
    print("❌ Please enter a valid number!")
```

🖥️ Output (if input = 0):

```
❌ You cannot divide by zero!
```

🧩 Explanation:

We can catch **specific exception types** to handle different errors differently.

🧩 Example 3: Multiple Exceptions Together

```
try:
    x = int(input("Enter X: "))
    y = int(input("Enter Y: "))
    print(x / y)
```

```
except (ZeroDivisionError, ValueError):
    print("⚠️ Error: Either division by zero or invalid input!")
```

🖥️ **Output (if y = 0):**

```
⚠️ Error: Either division by zero or invalid input!
```

🧩 Example 4: Catching All Exceptions

```
try:
    print(10 / 0)
except Exception as e:
    print("Error:", e)
```

🖥️ **Output:**

```
Error: division by zero
```

🧩 **Explanation:**
We can use `Exception as e` to **display the actual error message**.

🧱 5️⃣ Using `try`, `except`, `else`, and `finally`

💡 Concept Explanation

- **try:** code that may cause an exception
- **except:** runs if exception occurs
- **else:** runs if no exception occurs
- **finally:** runs no matter what (used for cleanup, closing files, etc.)

⚙️ Syntax

```
try:
    # Code that may raise error
except:
    # Code to handle the error
else:
    # Executes when no error
finally:
    # Always executes
```

🧩 Example 1: Try-Except-Else

```
try:
    num = int(input("Enter number: "))
    print("Square:", num ** 2)
except ValueError:
    print("❌ Invalid input!")
else:
    print("✅ No errors occurred!")
```

🖥️ Output (if input = 5):

```
Square: 25
✅ No errors occurred!
```

🧩 Explanation:

`else` block executes only if there is **no error** in the try block.

🧩 Example 2: Try-Except-Finally

```
try:
    f = open("data.txt", "r")
    print(f.read())
except FileNotFoundError:
    print("⚠️ File not found!")
finally:
    print("✅ Program execution completed.")
```

🖥️ Output (if file does not exist):

```
⚠️ File not found!
✅ Program execution completed.
```

🧩 Explanation:

Even if an error occurs, the `finally` block **always runs**.

🧩 Example 3: Full Try–Except–Else–Finally

```
try:
    a = int(input("Enter A: "))
    b = int(input("Enter B: "))
    result = a / b
except ZeroDivisionError:
    print("❌ Cannot divide by zero!")
else:
    print("Result =", result)
finally:
    print("✅ End of Program.")
```

🖥️ **Output (if a=10, b=2):**

```
Result = 5.0
✅ End of Program.
```

🧱 6️⃣ Raising Exceptions using `raise`

💡 Concept Explanation

Sometimes, you may want to **forcefully raise** an exception when certain conditions occur.
Use the `raise` keyword to generate exceptions manually.

⚙️ Syntax

```
raise ExceptionType("Custom error message")
```

🧩 Example 1: Raise ValueError

```
age = int(input("Enter your age: "))
if age < 0:
    raise ValueError("Age cannot be negative!")
else:
    print("Your age is", age)
```

🖥️ **Output (if age = -5):**

```
ValueError: Age cannot be negative!
```

🧩 **Explanation:**

The program stops and displays a **custom error message**.

🧩 Example 2: Custom Exception for Input Validation

```
marks = int(input("Enter marks: "))
if marks > 100:
    raise Exception("Marks cannot exceed 100!")
else:
    print("Marks recorded successfully!")
```

🖥️ **Output (if marks = 120):**

```
Exception: Marks cannot exceed 100!
```

🧱 7️⃣ Nested Try Blocks

💡 Concept Explanation

You can use **try blocks inside another try** block to handle specific sections of code separately.

🧩 Example

```
try:
    x = int(input("Enter a number: "))
    try:
        print("Result:", 10 / x)
    except ZeroDivisionError:
        print("⚠️ Division by zero not allowed!")
except ValueError:
    print("⚠️ Please enter a valid integer.")
```

🖥️ **Output (if x = 0):**

```
⚠️ Division by zero not allowed!
```

🧩 **Explanation:**
The inner try handles the division, while the outer try handles invalid input.

🧱 8️⃣ Common Exception Types

Exception Type	Description	Example
`ZeroDivisionError`	Dividing by zero	`10 / 0`
`ValueError`	Invalid value (e.g., converting text to int)	`int('abc')`
`FileNotFoundError`	File doesn’t exist	`open('abc.txt')`
`TypeError`	Wrong data type used	`5 + 'hi'`
`IndexError`	Invalid index in list	`a[5]`
`KeyError`	Missing key in dictionary	`d['name']`

🧱 9️⃣ Real-Life Example: File Handling

```
try:
    f = open("myfile.txt", "r")
    data = f.read()
    print(data)
except FileNotFoundError:
    print("❌ File not found.")
finally:
    print("✅ Closing file process complete.")
```

🖥️ **Output:**

```
❌ File not found.
✅ Closing file process complete.
```

🧱 🔟 Example: Division Calculator

```
try:
    a = int(input("Enter numerator: "))
    b = int(input("Enter denominator: "))
    result = a / b
except ZeroDivisionError:
    print("❌ Denominator cannot be zero!")
except ValueError:
    print("⚠️ Invalid number entered!")
else:
    print("Result:", result)
finally:
    print("✅ Program executed successfully.")
```

🖥️ **Output (if a=10, b=0):**

```
❌ Denominator cannot be zero!
✅ Program executed successfully.
```

🧩 1️⃣1️⃣ Exercises for Practice

1. Write a program to handle division by zero and invalid inputs.
2. Handle file not found error using `try` and `except`.
3. Use `try-except-else-finally` to divide two numbers safely.
4. Raise an exception if user enters a negative number.
5. Write a program that opens a file, reads data, and closes it safely.
6. Write a program to raise an exception if age > 150.

🧠 1️⃣2️⃣ Summary

Keyword	Description
`try`	Contains risky code that may cause an error
`except`	Handles the error when it occurs
`else`	Runs only when no error occurs
`finally`	Always runs (cleanup)
`raise`	Used to raise exceptions manually

✅ **Conclusion:**

Exception handling ensures your Python program doesn't crash unexpectedly.

It provides a safe, predictable way to handle errors and maintain program stability.

📚 Introduction to Python Libraries

🧩 What is a Library?

💡 Definition

A **library** in Python is a **collection of pre-written modules and functions** that make programming easier.
Instead of writing code from scratch, we can simply **import** these libraries and use their ready-made features.

🧠 Explanation

Python has thousands of libraries for various purposes:

- **Math operations** → `math` , `statistics`
- **File and OS handling** → `os` , `shutil`
- **Data analysis** → `pandas` , `numpy`
- **Visualization** → `matplotlib` , `seaborn`
- **Web development** → `flask` , `django`

⚙️ Example

```
import math

print(math.sqrt(25))      # Square root
print(math.pow(2, 3))     # Power
```

🖥️ **Output:**

```
5.0
8.0
```

🧩 **Explanation:**
The `math` library provides predefined mathematical functions such as `sqrt()` and `pow()` .

We just **import** and use them — no need to write the formulas ourselves!

🧩 1️⃣ Introduction to Pandas

💡 What is Pandas?

Pandas is a **powerful Python library** used for **data manipulation and analysis**.
It allows us to work with **tabular data** easily — similar to how we handle data in **spreadsheets or databases**.

🧠 Why Use Pandas?

Normally in Python, handling large data using lists or loops is time-consuming.
Pandas provides special data structures and built-in functions to:

- Read, write, and clean data
- Filter, sort, and group data
- Analyze and visualize data efficiently

⚙️ Installation and Import

If Pandas is not installed, install it first:

```
pip install pandas
```

Then import it in your Python program:

```
import pandas as pd
```

`pd` is the standard alias name used for Pandas.

📚 Example: Simple Use of Pandas

```
import pandas as pd
```

```
data = [10, 20, 30, 40]
s = pd.Series(data)
print(s)
```

🖥️ Output:

```
0     10
1     20
2     30
3     40
dtype: int64
```

🧩 Explanation:

- We created a simple **Series** using a Python list.
- Pandas automatically generated the **index (0, 1, 2, 3)** for each value.
- Each index-value pair represents one element of the Series.

🧱 2️⃣ Pandas Data Structures

Pandas provides two major data structures:

Data Structure	Dimension	Description
Series	1D	A single column of data with labels
DataFrame	2D	A table with rows and columns

🧩 3️⃣ Series in Pandas

💡 What is a Series?

A **Series** is a **one-dimensional labeled array** that can hold any data type such as integers, floats, or strings.

Think of a **Series** like a single **column in an Excel sheet** — it has values and labels (index).

⚙️ Syntax

```
pandas.Series(data, index, dtype, name)
```

Parameter	Description
`data`	The data (list, array, dictionary, etc.)
`index`	Custom labels for data
`dtype`	Data type of values
`name`	Optional name for the Series

🧩 Example 1: Create Series from a List

```
import pandas as pd

numbers = [10, 20, 30, 40]
s = pd.Series(numbers)
print(s)
```

🖥️ Output:

```
0     10
1     20
2     30
3     40
dtype: int64
```

🧩 Explanation:

- Here, `[10, 20, 30, 40]` is the list converted into a Pandas Series.
- The index (0, 1, 2, 3) is automatically assigned.
- The `dtype` shows the data type as integer.

🧩 Example 2: Create Series with Custom Index

```
marks = [85, 90, 78, 92]
students = ['John', 'Sara', 'Ali', 'Priya']

series = pd.Series(marks, index=students)
print(series)
```

🖥️ **Output:**

```
John      85
Sara      90
Ali       78
Priya     92
dtype: int64
```

🧩 **Explanation:**

- Each mark is associated with a **student name**.
- You can now access data using **index labels** like `'Sara'` instead of numeric indexes.

🧩 Example 3: Create Series from a Dictionary

```
data = {'a': 10, 'b': 20, 'c': 30}
s = pd.Series(data)
print(s)
```

🖥️ **Output:**

```
a     10
b     20
c     30
dtype: int64
```

🧩 **Explanation:**

- The keys become **indexes** and values become **data**.

🧩 Example 4: Accessing Elements

```
print(series['Sara'])
print(series[2])
```

🖥️ **Output:**

```
90
78
```

🧩 **Explanation:**

- You can access elements either by **index label ('Sara')** or by **numeric position (2)**.

🧩 Example 5: Performing Operations on Series

```
print(series + 5)        # Adds 5 to all values
print(series.mean())     # Calculates average marks
print(series.max())      # Finds highest marks
```

🖥️ **Output:**

```
John      90
Sara      95
Ali       83
Priya     97
dtype: int64

Average Marks: 86.25
Highest Marks: 92
```

🧩 **Explanation:**

- Pandas performs **vectorized operations** automatically on all elements without using loops.

🧩 Useful Series Functions

Function	Description	Example
`series.head()`	Shows first 5 elements	`series.head()`
`series.tail()`	Shows last 5 elements	`series.tail()`
`series.sum()`	Sum of all elements	`series.sum()`
`series.mean()`	Average of all elements	`series.mean()`
`series.sort_values()`	Sorts the Series	`series.sort_values()`

🧱 4️⃣ DataFrame in Pandas

💡 What is a DataFrame?

A **DataFrame** is a **two-dimensional labeled data structure**.
It is similar to a table or an Excel spreadsheet — containing **rows and columns**.

Each column in a DataFrame is a **Series**.

⚙️ Syntax

```
pandas.DataFrame(data, index, columns, dtype)
```

Parameter	Description
`data`	Data (list, dict, array, Series, etc.)
`index`	Row labels
`columns`	Column labels
`dtype`	Data type of the elements

🧩 Example 1: Create DataFrame from Dictionary

```
import pandas as pd

data = {
    'Name': ['John', 'Sara', 'Ali', 'Priya'],
    'Age': [21, 20, 22, 19],
    'Marks': [85, 90, 78, 92]
}

df = pd.DataFrame(data)
print(df)
```

🖥️ **Output:**

```
    Name  Age  Marks
0   John   21     85
1   Sara   20     90
2    Ali   22     78
3  Priya   19     92
```

🧩 **Explanation:**

- Each **key** in the dictionary becomes a **column**.
- Each **value list** becomes the **data** in that column.
- Pandas automatically assigns indexes (0–3).

🧩 Example 2: Access Columns and Rows

```
print(df['Name'])              # Access single column
print(df[['Name', 'Marks']])   # Access multiple columns
print(df.iloc[0])              # Access first row
print(df.loc[2, 'Marks'])      # Access specific cell
```

🖥️ **Output:**

```
0     John
1     Sara
2      Ali
3    Priya
Name: Name, dtype: object

    Name  Marks
0   John     85
1   Sara     90
2    Ali     78
3  Priya     92

Name      John
Age         21
Marks       85
Name: 0, dtype: object

78
```

🧩 **Explanation:**

- `iloc[]` → integer location based (by position).
- `loc[]` → label-based access (by index name or column label).

🧩 Example 3: Adding and Removing Columns

```
df['Result'] = ['Pass', 'Pass', 'Fail', 'Pass']
print(df)

df = df.drop('Result', axis=1)
print(df)
```

🖥️ **Output:**

```
    Name  Age  Marks Result
0   John   21     85   Pass
1   Sara   20     90   Pass
2    Ali   22     78   Fail
3  Priya   19     92   Pass

    Name  Age  Marks
0   John   21     85
1   Sara   20     90
2    Ali   22     78
3  Priya   19     92
```

🧩 **Explanation:**

- New columns can be added easily.
- Use `drop(column, axis=1)` to remove a column.

🧩 Example 4: Basic Operations

```
print(df.head())         # First 5 rows
print(df.shape)          # (rows, columns)
print(df.describe())     # Statistics summary
print(df.columns)        # Column names
print(df.dtypes)         # Data types
```

🖥️ **Output (partial):**

```
    Name  Age  Marks
0   John   21     85
1   Sara   20     90
2    Ali   22     78
3  Priya   19     92
```

```
(4, 3)
        Age      Marks
count   4.0   4.000000
mean   20.5   86.25
std     1.29    6.19
min    19.0   78.0
max    22.0   92.0
```

🧩 Explanation:

- `head()` gives a preview of data.
- `shape` shows how big the dataset is.
- `describe()` gives useful numerical stats.

🧩 Example 5: Filtering and Sorting Data

```
high_scorers = df[df['Marks'] > 80]
print(high_scorers)

print(df.sort_values('Marks'))
```

🖥️ Output:

```
    Name  Age  Marks
0   John   21     85
1   Sara   20     90
3  Priya   19     92

    Name  Age  Marks
2    Ali   22     78
0   John   21     85
1   Sara   20     90
3  Priya   19     92
```

🧩 Explanation:

- You can **filter rows** using conditions.
- `sort_values()` arranges data in ascending order by default.

🧩 5️⃣ Reading and Writing CSV Files

💡 What is a CSV File?

A **CSV (Comma-Separated Values)** file is a simple text file used to store tabular data, separated by commas.

Example CSV content:

```
Name,Age,Marks
John,21,85
Sara,20,90
Ali,22,78
Priya,19,92
```

⚙️ Reading CSV Files

```
import pandas as pd

df = pd.read_csv('students.csv')
print(df)
```

🖥️ **Output:**

```
    Name  Age  Marks
0   John   21     85
1   Sara   20     90
2    Ali   22     78
3  Priya   19     92
```

🧩 **Explanation:**

- `read_csv()` reads the CSV file and converts it into a **DataFrame**.

⚙️ Writing DataFrame to CSV File

```
df.to_csv('output.csv', index=False)
print("✅ Data saved successfully!")
```

🧩 **Explanation:**

- `to_csv()` saves your DataFrame to a file.
- `index=False` avoids saving row numbers.

🧩 Example: Display Top Rows and Statistics

```
print(df.head(2))          # First 2 rows
print(df.describe())       # Summary statistics
```

🖥️ **Output:**

```
   Name   Age   Marks
0  John    21      85
1  Sara    20      90

        Age       Marks
count    4.0    4.000000
mean    20.5   86.25
std      1.29    6.19
min     19.0   78.0
max     22.0   92.0
```

🧩 6️⃣ Exercises for Practice

1. Create a Pandas Series for 5 products and their prices.
2. Create a DataFrame of students (Name, Age, Marks).
3. Add a new column `Result` = "Pass" if Marks > 40 else "Fail".
4. Read a CSV file named `data.csv` and display first 3 rows.
5. Write your DataFrame to `result.csv`.
6. Sort DataFrame by `Marks` in descending order.
7. Display average, minimum, and maximum marks.

🧠 Summary

Concept	Description
Series	1D labeled array
DataFrame	2D labeled table
read_csv()	Reads CSV data
to_csv()	Writes CSV data
head(), tail()	Show data preview
describe()	Show statistics summary
filtering	Select specific data
sorting	Arrange data in order