Problem Solving Using Python

Unit III: Exception Handling and Libraries

Course Code: 25ENUEC158

Credits: 4

Course Type: IPCC

Prepared by:

Mohammed Siraj B

School of Engineering
St Aloysius (Deemed to be University), Mangalore

For B.Tech - I/II Semester
Academic Year 2024-25

Contents

1 Introduction to Exceptions and Errors

1.1 What are Exceptions?

Definition

An **exception** is an error that occurs during the execution of a program. When Python encounters an error, it stops the program and displays an error message. This process is called "raising an exception."

1.2 Simplified Explanation

Think of exceptions like unexpected problems in real life:

- You try to open a door, but the key doesn't work (FileNotFoundError)
- You try to divide something into zero parts (ZeroDivisionError)
- You try to add a number to text (TypeError)

Instead of your program crashing completely, Python gives us tools to handle these problems gracefully!

1.3 Common Types of Exceptions

Exception Type	When it Occurs
`ZeroDivisionError`	When dividing a number by zero
`ValueError`	When an operation receives an inappropriate value
`TypeError`	When an operation is applied to wrong data type
`FileNotFoundError`	When trying to open a file that doesn't exist
`IndexError`	When trying to access an invalid index in a list
`KeyError`	When trying to access a key that doesn't exist in dictionary
`NameError`	When using a variable that hasn't been defined

Table 1: Common Python Exceptions

1.4 Example 1: Understanding Basic Exceptions

Example 1: Division by Zero

```
# This program will crash!
print("Welcome to Division Program")
a = 10
b = 0
result = a / b  # This will raise ZeroDivisionError
print("Result:", result)
```

Output

```
Welcome to Division Program
Traceback (most recent call last):
  File "program.py", line 4, in <module>
    result = a / b
ZeroDivisionError: division by zero
```

1.5 Example 2: Type Error

Example 2: Adding Number and String

```
# This will also crash!
age = 25
name = "John"
result = age + name  # Can't add number and string
print(result)
```

Output

```
Traceback (most recent call last):
  File "program.py", line 3, in <module>
    result = age + name
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

1.6 Example 3: Index Error

Example 3: Accessing Invalid List Index

```
# Trying to access element that doesn't exist
fruits = ["apple", "banana", "orange"]
print(fruits[0])  # This works fine
print(fruits[5])  # This will crash - only 3 elements
    exist!
```

Output

```
apple
Traceback (most recent call last):
  File "program.py", line 3, in <module>
    print(fruits[5])
IndexError: list index out of range
```

Important Note

When an exception occurs, Python stops executing the program immediately. The lines after the error will NOT run. This is why we need exception handling!

2 Exception Handling using Try-Except

2.1 What is Exception Handling?

Definition

Exception Handling is a mechanism to handle runtime errors gracefully so that the program can continue running or exit smoothly instead of crashing abruptly.

2.2 General Syntax

```
try:
    # Code that might raise an exception
    # Risky operations go here
except ExceptionType:
    # Code to handle the exception
    # This runs if exception occurs
```

2.3 How it Works

1. Python tries to execute the code in the `try` block
2. If an error occurs, Python immediately jumps to the `except` block
3. If no error occurs, the `except` block is skipped
4. The program continues running after the try-except block

2.4 Example 4: Basic Try-Except

Example 4: Handling Division by Zero

```
# Safe division program
print("Welcome to Safe Division Program")

try:
    a = 10
    b = 0
    result = a / b
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
    print("Please use a non-zero number.")

print("Program continues running...")
```

Output

```
Welcome to Safe Division Program
Error: Cannot divide by zero!
Please use a non-zero number.
Program continues running...
```

2.5 Example 5: Handling User Input

Example 5: Safe User Input

```python
# Getting age from user safely
print("Age Calculator")

try:
    age = int(input("Enter your age: "))
    print("Your age is:", age)
    years_to_hundred = 100 - age
    print("Years to reach 100:", years_to_hundred)
except ValueError:
    print("Error: Please enter a valid number!")
    print("You entered text instead of a number.")

print("Thank you for using the calculator!")
```

Output

```
If user enters "twenty":
Age Calculator
Enter your age: twenty
Error: Please enter a valid number!
You entered text instead of a number.
Thank you for using the calculator!

If user enters 25:
Age Calculator
Enter your age: 25
Your age is: 25
Years to reach 100: 75
Thank you for using the calculator!
```

2.6 Example 6: Multiple Except Blocks

Example 6: Handling Different Exceptions

```python
# Calculator with multiple error handling
print("Simple Calculator")

try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))
    result = num1 / num2
    print("Result:", result)

except ValueError:
    print("Error: Please enter valid numbers only!")

except ZeroDivisionError:
    print("Error: Second number cannot be zero!")

print("Calculator closed.")
```

Output

```
Test Case 1 (Invalid input):
Simple Calculator
Enter first number: abc
Error: Please enter valid numbers only!
Calculator closed.

Test Case 2 (Division by zero):
Simple Calculator
Enter first number: 10
Enter second number: 0
Error: Second number cannot be zero!
Calculator closed.

Test Case 3 (Success):
Simple Calculator
Enter first number: 10
Enter second number: 2
Result: 5.0
Calculator closed.
```

2.7 Example 7: Generic Exception Handling

Example 7: Catching All Exceptions

```
# Catching any type of error
print("List Access Program")

numbers = [10, 20, 30, 40, 50]

try:
    index = int(input("Enter index to access: "))
    print("Value at index", index, "is:", numbers[index])

except Exception as e:
    print("An error occurred!")
    print("Error details:", e)
    print("Please try again with a valid index (0-4)")

print("Program ended.")
```

Output

```
Test Case 1:
List Access Program
Enter index to access: 10
An error occurred!
Error details: list index out of range
Please try again with a valid index (0-4)
Program ended.

Test Case 2:
List Access Program
Enter index to access: 2
Value at index 2 is: 30
Program ended.
```

Important Note

Using `Exception` catches all types of errors. The `as e` part captures the error message so you can display it or log it.

3 Try-Except-Else-Finally

3.1 Extended Exception Handling

Definition

Python provides additional clauses for exception handling:

- **else:** Runs only if no exception occurred in try block
- **finally:** Always runs, whether exception occurred or not

3.2 Complete Syntax

```
try:
    # Code that might raise an exception

except ExceptionType:
    # Handles specific exception

else:
    # Runs only if no exception occurred

finally:
    # Always runs (cleanup code)
```

3.3 Execution Flow

Block	When Exception Occurs	When No Exception
`try`	Executes until error	Executes completely
`except`	Executes	Does NOT execute
`else`	Does NOT execute	Executes
`finally`	Always executes	Always executes

Table 2: Exception Handling Flow

3.4 Example 8: Using Else Block

Example 8: Division with Else

```python
# Division program with else
print("Division Calculator")

try:
    a = int(input("Enter first number: "))
    b = int(input("Enter second number: "))
    result = a / b

except ZeroDivisionError:
    print("Cannot divide by zero!")

except ValueError:
    print("Please enter valid numbers!")

else:
    # This runs only if no exception occurred
    print("Division successful!")
    print("Result:", result)

print("Program completed.")
```

Output

```
Test Case 1 (Success):
Division Calculator
Enter first number: 20
Enter second number: 4
Division successful!
Result: 5.0
Program completed.

Test Case 2 (Error):
Division Calculator
Enter first number: 20
Enter second number: 0
Cannot divide by zero!
Program completed.
```

3.5 Example 9: Using Finally Block

Example 9: File Operations with Finally

```python
# Reading a file safely
print("File Reader Program")

try:
    file = open("data.txt", "r")
    content = file.read()
    print("File Content:", content)

except FileNotFoundError:
    print("Error: File not found!")

else:
    print("File read successfully!")

finally:
    # This always runs - close file if it was opened
    print("Cleanup: Closing file operations...")
    try:
        file.close()
        print("File closed.")
    except:
        print("No file to close.")

print("Program ended.")
```

Output

```
If file exists:
File Reader Program
File Content: Hello World! This is a test file.
File read successfully!
Cleanup: Closing file operations...
File closed.
Program ended.

If file doesn't exist:
File Reader Program
Error: File not found!
Cleanup: Closing file operations...
No file to close.
Program ended.
```

3.6 Example 10: Complete Exception Handling

Example 10: Bank Account Withdrawal

```python
# Bank withdrawal system
print("=== Bank Withdrawal System ===")

balance = 5000

try:
    print("Current Balance: Rs.", balance)
    amount = float(input("Enter withdrawal amount: "))

    if amount <= 0:
        raise ValueError("Amount must be positive!")

    if amount > balance:
        raise ValueError("Insufficient balance!")

    # Process withdrawal
    balance = balance - amount

except ValueError as e:
    print("Error:", e)

else:
    # Success case
    print("\nWithdrawal successful!")
    print("Amount withdrawn: Rs.", amount)
    print("Remaining balance: Rs.", balance)

finally:
    # Always print transaction record
    print("\n--- Transaction Complete ---")
    print("Thank you for using our service!")
```

Output

```
Test Case 1 (Success):
=== Bank Withdrawal System ===
Current Balance: Rs. 5000
Enter withdrawal amount: 2000

Withdrawal successful!
Amount withdrawn: Rs. 2000.0
Remaining balance: Rs. 3000.0

--- Transaction Complete ---
Thank you for using our service!

Test Case 2 (Insufficient balance):
=== Bank Withdrawal System ===
Current Balance: Rs. 5000
Enter withdrawal amount: 7000
Error: Insufficient balance!

--- Transaction Complete ---
Thank you for using our service!
```

Important Note

The `finally` block is perfect for cleanup operations like:

- Closing files
- Closing database connections
- Logging transactions
- Releasing resources

4 Introduction to Pandas

4.1 What is Pandas?

Definition

Pandas is a powerful Python library used for data analysis and manipulation. It provides easy-to-use data structures for working with structured data (like tables).

4.2 Why Use Pandas?

- Work with data in table format (like Excel)
- Read data from various file formats (CSV, Excel, etc.)
- Clean and prepare data for analysis
- Perform calculations and statistical operations
- Filter, sort, and group data easily

4.3 Installing Pandas

```
# Install pandas using pip
pip install pandas
```

4.4 Importing Pandas

```
# Standard way to import pandas
import pandas as pd

# 'pd' is the standard abbreviation for pandas
```

4.5 Main Data Structures in Pandas

Structure	Description
Series	One-dimensional array-like object (like a single column)
DataFrame	Two-dimensional table with rows and columns (like Excel sheet)

Table 3: Pandas Data Structures

5 Pandas Series

5.1 What is a Series?

Definition

A **Series** is a one-dimensional labeled array that can hold any data type (integers, strings, floats, etc.). Think of it as a single column in a spreadsheet.

5.2 Creating a Series - General Syntax

```
import pandas as pd

# Syntax
series_name = pd.Series(data, index=labels)

# data: list, array, or dictionary
# index: labels for each element (optional)
```

5.3 Example 11: Creating Basic Series

Example 11: Simple Series

```
import pandas as pd

# Creating a Series from a list
marks = pd.Series([85, 90, 78, 92, 88])

print("Student Marks:")
print(marks)
print("\nType:", type(marks))
```

Output

```
Student Marks:
0    85
1    90
2    78
3    92
4    88
dtype: int64

Type: <class 'pandas.core.series.Series'>
```

5.4 Example 12: Series with Custom Index

Example 12: Series with Labels

```python
import pandas as pd

# Creating Series with custom labels
subjects = ['Math', 'Physics', 'Chemistry', 'Biology', 'English']
marks = [85, 90, 78, 92, 88]

marks_series = pd.Series(marks, index=subjects)

print("Subject-wise Marks:")
print(marks_series)

print("\n--- Accessing Individual Values ---")
print("Math marks:", marks_series['Math'])
print("Biology marks:", marks_series['Biology'])
```

Output

```
Subject-wise Marks:
Math         85
Physics      90
Chemistry    78
Biology      92
English      88
dtype: int64

--- Accessing Individual Values ---
Math marks: 85
Biology marks: 92
```

5.5 Example 13: Series Operations

Example 13: Mathematical Operations

```
import pandas as pd

# Creating a Series
prices = pd.Series([100, 200, 150, 300, 250],
                   index=['Item1', 'Item2', 'Item3', '
                      Item4', 'Item5'])

print("Original Prices:")
print(prices)

# Applying 10% discount
discounted_prices = prices * 0.9

print("\nAfter 10% Discount:")
print(discounted_prices)

# Adding tax (5%)
final_prices = discounted_prices * 1.05

print("\nFinal Prices (with 5% tax):")
print(final_prices)

# Statistics
print("\n--- Statistics ---")
print("Maximum price:", prices.max())
print("Minimum price:", prices.min())
print("Average price:", prices.mean())
print("Total:", prices.sum())
```

Output

```
Original Prices:
Item1    100
Item2    200
Item3    150
Item4    300
Item5    250
dtype: int64

After 10% Discount:
Item1     90.0
Item2    180.0
Item3    135.0
Item4    270.0
Item5    225.0
dtype: float64

Final Prices (with 5% tax):
Item1     94.50
Item2    189.00
Item3    141.75
Item4    283.50
Item5    236.25
dtype: float64

--- Statistics ---
Maximum price: 300
Minimum price: 100
Average price: 200.0
Total: 1000
```

6 Pandas DataFrame

6.1 What is a DataFrame?

Definition

A **DataFrame** is a two-dimensional labeled data structure with columns that can be of different types. It's like a spreadsheet or SQL table - data is arranged in rows and columns.

6.2 Creating a DataFrame - General Syntax

```
import pandas as pd

# Method 1: From dictionary
df = pd.DataFrame(dictionary)

# Method 2: From lists
df = pd.DataFrame(list_of_lists, columns=column_names)

# Method 3: From CSV file
df = pd.read_csv('filename.csv')
```

6.3 Example 14: Creating DataFrame from Dictionary

Example 14: Student Records

```
import pandas as pd

# Creating DataFrame from dictionary
student_data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Age': [20, 21, 19, 22, 20],
    'Marks': [85, 90, 78, 92, 88],
    'Grade': ['B', 'A', 'C', 'A', 'B']
}

df = pd.DataFrame(student_data)

print("Student DataFrame:")
print(df)

print("\n--- DataFrame Information ---")
print("Shape (rows, columns):", df.shape)
print("Column names:", df.columns.tolist())
print("Data types:")
print(df.dtypes)
```

Output

```
Student DataFrame:
      Name  Age  Marks Grade
0    Alice   20     85     B
1      Bob   21     90     A
2  Charlie   19     78     C
3    David   22     92     A
4      Eva   20     88     B

--- DataFrame Information ---
Shape (rows, columns): (5, 4)
Column names: ['Name', 'Age', 'Marks', 'Grade']
Data types:
Name     object
Age       int64
Marks     int64
Grade    object
dtype: object
```

6.4 Example 15: Accessing DataFrame Data

Example 15: Selecting Data

```
import pandas as pd

# Creating DataFrame
data = {
    'Product': ['Laptop', 'Mouse', 'Keyboard', 'Monitor',
       'Printer'],
    'Price': [50000, 500, 1500, 15000, 8000],
    'Stock': [10, 50, 30, 15, 20]
}

df = pd.DataFrame(data)

print("Complete DataFrame:")
print(df)

print("\n--- Selecting Single Column ---")
print(df['Product'])

print("\n--- Selecting Multiple Columns ---")
print(df[['Product', 'Price']])

print("\n--- Selecting Rows (first 3) ---")
print(df.head(3))

print("\n--- Selecting Specific Row (index 2) ---")
print(df.iloc[2])

print("\n--- Filtering (Price > 5000) ---")
expensive_items = df[df['Price'] > 5000]
print(expensive_items)
```

Output

```
Complete DataFrame:
    Product  Price  Stock
0    Laptop  50000     10
1     Mouse    500     50
2  Keyboard   1500     30
3   Monitor  15000     15
4   Printer   8000     20

--- Selecting Single Column ---
0      Laptop
1       Mouse
2    Keyboard
3     Monitor
4     Printer
Name: Product, dtype: object

--- Selecting Multiple Columns ---
    Product  Price
0    Laptop  50000
1     Mouse    500
2  Keyboard   1500
3   Monitor  15000
4   Printer   8000

--- Selecting Rows (first 3) ---
    Product  Price  Stock
0    Laptop  50000     10
1     Mouse    500     50
2  Keyboard   1500     30

--- Selecting Specific Row (index 2) ---
Product    Keyboard
Price          1500
Stock            30
Name: 2, dtype: object

--- Filtering (Price > 5000) ---
   Product  Price  Stock
0   Laptop  50000     10
3  Monitor  15000     15
4  Printer   8000     20
```

6.5 Example 16: Basic DataFrame Operations

Example 16: Data Analysis

```
import pandas as pd

# Creating sales DataFrame
sales_data = {
    'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
    'Sales': [25000, 28000, 32000, 30000, 35000],
    'Expenses': [15000, 16000, 18000, 17000, 19000]
}

df = pd.DataFrame(sales_data)

print("Sales Data:")
print(df)

# Adding new column (Profit)
df['Profit'] = df['Sales'] - df['Expenses']

print("\nAfter adding Profit column:")
print(df)

# Statistics
print("\n--- Statistical Summary ---")
print(df.describe())

print("\n--- Specific Statistics ---")
print("Total Sales:", df['Sales'].sum())
print("Average Profit:", df['Profit'].mean())
print("Maximum Sales:", df['Sales'].max())
print("Month with Max Sales:", df[df['Sales'] == df['Sales
    '].max()]['Month'].values[0])
```

Output

```
Sales Data:
  Month  Sales  Expenses
0   Jan  25000     15000
1   Feb  28000     16000
2   Mar  32000     18000
3   Apr  30000     17000
4   May  35000     19000

After adding Profit column:
  Month  Sales  Expenses  Profit
0   Jan  25000     15000   10000
1   Feb  28000     16000   12000
2   Mar  32000     18000   14000
3   Apr  30000     17000   13000
4   May  35000     19000   16000

--- Statistical Summary ---
           Sales     Expenses       Profit
count      5.000        5.000        5.000
mean   30000.000    17000.000    13000.000
std     3807.887     1581.139     2345.208
min    25000.000    15000.000    10000.000
25%    28000.000    16000.000    12000.000
50%    30000.000    17000.000    13000.000
75%    32000.000    18000.000    14000.000
max    35000.000    19000.000    16000.000

--- Specific Statistics ---
Total Sales: 150000
Average Profit: 13000.0
Maximum Sales: 35000
Month with Max Sales: May
```

7 Reading CSV Files Using Pandas

7.1 What is a CSV File?

Definition

CSV stands for Comma-Separated Values. It's a simple file format used to store tabular data. Each line represents a row, and values are separated by commas.

7.2 General Syntax for Reading CSV

```
import pandas as pd

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

# Common parameters:
# sep=',' : separator (default is comma)
# header=0 : row number to use as column names
# index_col : column to use as row labels
# na_values : values to recognize as NaN
```

7.3 Example 17: Reading and Analyzing CSV

Example 17: Reading Student Data from CSV

First, let's create a CSV file:

```
import pandas as pd

# Creating sample data
data = {
    'StudentID': [101, 102, 103, 104, 105],
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Math': [85, 90, 78, 92, 88],
    'Science': [88, 85, 90, 95, 87],
    'English': [82, 88, 85, 90, 92]
}

df = pd.DataFrame(data)

# Save to CSV
df.to_csv('students.csv', index=False)
print("CSV file created: students.csv")

# Now read the CSV
print("\n--- Reading CSV File ---")
df_read = pd.read_csv('students.csv')

print("\nData from CSV:")
print(df_read)

print("\n--- Basic Analysis ---")
print("Number of students:", len(df_read))
print("\nAverage Marks per Subject:")
print("Math:", df_read['Math'].mean())
print("Science:", df_read['Science'].mean())
print("English:", df_read['English'].mean())

# Calculate total marks for each student
df_read['Total'] = df_read['Math'] + df_read['Science'] +
    df_read['English']
df_read['Average'] = df_read['Total'] / 3

print("\nFinal Report Card:")
print(df_read)
```

Output

```
CSV file created: students.csv

--- Reading CSV File ---

Data from CSV:
   StudentID     Name  Math  Science  English
0        101    Alice    85       88       82
1        102      Bob    90       85       88
2        103  Charlie    78       90       85
3        104    David    92       95       90
4        105      Eva    88       87       92

--- Basic Analysis ---
Number of students: 5

Average Marks per Subject:
Math: 86.6
Science: 89.0
English: 87.4

Final Report Card:
   StudentID     Name  Math  Science  English  Total     Average
0        101    Alice    85       88       82    255   85.000000
1        102      Bob    90       85       88    263   87.666667
2        103  Charlie    78       90       85    253   84.333333
3        104    David    92       95       90    277   92.333333
4        105      Eva    88       87       92    267   89.000000
```

7.4 Example 18: Data Cleaning from CSV

Example 18: Handling Missing Data

```python
import pandas as pd
import numpy as np

# Creating data with missing values
data = {
    'Product': ['Laptop', 'Mouse', 'Keyboard', 'Monitor',
       'Printer'],
    'Price': [50000, 500, None, 15000, 8000],
    'Rating': [4.5, None, 4.0, 4.8, 4.2],
    'InStock': ['Yes', 'Yes', 'No', 'Yes', 'Yes']
}

df = pd.DataFrame(data)
df.to_csv('products.csv', index=False)

# Read CSV
df = pd.read_csv('products.csv')

print("Original Data (with missing values):")
print(df)

# Check for missing values
print("\n--- Missing Values Check ---")
print(df.isnull().sum())

# Fill missing values
df['Price'].fillna(df['Price'].mean(), inplace=True)
df['Rating'].fillna(df['Rating'].mean(), inplace=True)

print("\nData After Filling Missing Values:")
print(df)

# Save cleaned data
df.to_csv('products_cleaned.csv', index=False)
print("\nCleaned data saved to: products_cleaned.csv")
```

Output

```
Original Data (with missing values):
    Product    Price  Rating InStock
0    Laptop  50000.0     4.5     Yes
1     Mouse    500.0     NaN     Yes
2  Keyboard      NaN     4.0      No
3   Monitor  15000.0     4.8     Yes
4   Printer   8000.0     4.2     Yes

--- Missing Values Check ---
Product     0
Price       1
Rating      1
InStock     0
dtype: int64

Data After Filling Missing Values:
    Product    Price  Rating InStock
0    Laptop  50000.0    4.50     Yes
1     Mouse    500.0    4.38     Yes
2  Keyboard  18375.0    4.00      No
3   Monitor  15000.0    4.80     Yes
4   Printer   8000.0    4.20     Yes

Cleaned data saved to: products_cleaned.csv
```

7.5 Common DataFrame Methods

Method	Description
`df.head(n)`	Show first n rows (default 5)
`df.tail(n)`	Show last n rows (default 5)
`df.info()`	Display DataFrame summary
`df.describe()`	Statistical summary of numeric columns
`df.shape`	Number of rows and columns
`df.columns`	List of column names
`df.isnull()`	Check for missing values
`df.fillna(value)`	Fill missing values
`df.dropna()`	Remove rows with missing values
`df.sort_values(by)`	Sort by column(s)
`df.groupby()`	Group data by column

Table 4: Useful DataFrame Methods

8 Practice Exercises

Practice Exercises

Exception Handling Exercises:

1. Write a program that asks the user for two numbers and divides them. Handle both `ValueError` (if user enters non-numeric input) and `ZeroDivisionError`.
2. Create a program that reads a file name from the user and displays its contents. Handle the case when the file doesn't exist.
3. Write a program to access list elements by index. Take index as user input and handle `IndexError` gracefully.
4. Create a dictionary of student marks. Write a program that asks for a student name and displays their marks. Handle `KeyError` if the name doesn't exist.
5. Write a calculator program that performs +, -, *, / operations. Handle all possible exceptions and use the `finally` block to print "Calculation completed."

Pandas Series Exercises:

6. Create a Series containing temperatures of 7 days. Calculate the average, maximum, and minimum temperature.
7. Create a Series of 5 product prices. Apply a 20% discount to all products and display the new prices.
8. Create a Series with student names as index and their marks as values. Find the student with highest marks.

Pandas DataFrame Exercises:

9. Create a DataFrame with columns: EmployeeID, Name, Salary, Department for 5 employees. Display only employees with Salary ¿ 30000.
10. Create a DataFrame of 5 products with columns: ProductName, Price, Quantity. Calculate the total value (Price × Quantity) for each product.
11. Create a DataFrame with student information (Name, Math, Physics, Chemistry marks). Calculate total marks and percentage for each student.
12. Create a CSV file containing monthly expenses (Month, Rent, Food, Transport). Read the file and calculate total expenses per month and category-wise total.

Advanced Exercises:

13. Create a program that reads student marks from a CSV file, handles any missing values, and generates a report showing class average, highest scorer, and lowest scorer.
14. Write a program that takes temperature data from a CSV file. Use exception handling for file operations and Pandas for data analysis. Display statistics and plot the data (if possible).
15. Create an inventory management system using DataFrame. Include functions to:

9 Quick Reference Guide

9.1 Exception Handling Cheat Sheet

Try-Except Patterns

```python
# Basic
try:
    risky_code()
except ExceptionType:
    handle_error()

# Multiple exceptions
try:
    risky_code()
except ValueError:
    handle_value_error()
except ZeroDivisionError:
    handle_zero_division()

# With else and finally
try:
    risky_code()
except Exception:
    handle_error()
else:
    success_code()
finally:
    cleanup_code()

# Generic exception
try:
    risky_code()
except Exception as e:
    print("Error:", e)
```

9.2 Pandas Quick Reference

Essential Pandas Commands

```python
import pandas as pd

# Creating Series
s = pd.Series([1, 2, 3])

# Creating DataFrame
df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})

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

# Writing CSV
df.to_csv('file.csv', index=False)

# Viewing data
df.head()       # First 5 rows
df.tail()       # Last 5 rows
df.info()       # DataFrame info
df.describe()   # Statistics

# Selecting
df['column']             # Single column
df[['col1', 'col2']]   # Multiple columns
df.iloc[0]             # Row by position
df.loc[0]              # Row by label

# Filtering
df[df['column'] > 5]

# Adding column
df['new_col'] = df['col1'] + df['col2']

# Statistics
df['column'].mean()
df['column'].sum()
df['column'].max()
df['column'].min()
```

9.3 Important Tips

Important Note

Exception Handling Best Practices:

- Always handle specific exceptions first, then generic ones
- Use meaningful error messages
- Don't catch exceptions you can't handle
- Always clean up resources in finally block
- Log errors for debugging

Pandas Best Practices:

- Always use `index=False` when saving to CSV to avoid extra column
- Check for missing values before analysis
- Use descriptive column names
- Verify data types after reading files
- Keep backups of original data

End of Unit III Notes

Happy Learning! Practice regularly to master these concepts.

Prepared with care for beginner students