**DATA ENGINEERING**

**PYTHON ASSIGNMENT**

**1. Data Manipulation with Pandas:**

**1a. Given a dataset in the form of a dictionary, convert it to a DataFrame and perform**

**the following tasks:**

import pandas as pd

data = {

'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],

'Age': [25, 30, None, 35, 40], # introducing missing value for demonstration

'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Seattle'],

'Gender': ['F', 'M', 'M', 'M', 'F']

}

df = pd.DataFrame(data)

print("Initial DataFrame:")

print(df)

**1. Remove rows with missing values.**

Ans:

df\_cleaned = df.dropna()

print("\nDataFrame after removing rows with missing values:")

print(df\_cleaned)

**2. Group the data by 'Product' and calculate the total sales for each product.**

Ans:

total\_sales = df.groupby('Product')['Sales'].sum().reset\_index()

print("\nTotal sales for each product:")

print(total\_sales)

**3.Sort the results by total sales in descending order**

Ans:

total\_sales\_sorted = total\_sales.sort\_values(by='Sales', ascending=False)

print("\nTotal sales sorted by total sales in descending order:")

print(total\_sales\_sorted)

**1b.Create a pivot table that shows the sum of sales for each product, broken down**

**by month.**

Ans:

import pandas as pd

data = {

'Product': ['A', 'B', 'A', 'B', 'A'],

'Sales': [100, 150, 200, 300, 250],

'Month': ['Jan', 'Feb', 'Jan', 'Feb', 'Mar']

}

df = pd.DataFrame(data)

print("Initial DataFrame:")

print(df)

pivot\_table = pd.pivot\_table(df, values='Sales', index='Product', columns='Month', aggfunc='sum')

print("\nPivot table showing sum of sales for each product, broken down by month:")

print(pivot\_table)

**2. Data Cleaning:**

**2a.Write a function that takes a DataFrame with various types of data (numeric,**

**text, dates) and performs the following cleaning steps:**

Ans:

import pandas as pd

import numpy as np

**1.Replace all empty strings with NaN.**

Ans:

def clean\_dataframe(df):

df.replace('', np.nan, inplace=True)

**2.Fill numeric NaNs with the mean of their column.**

Ans:

for col in df.select\_dtypes(include='number'):

df[col].fillna(df[col].mean(), inplace=True)

**3.Convert all text to lowercase.**

Ans:

df = df.apply(lambda x: x.str.lower() if x.dtype == 'object' else x)

return df

data = {

'Name': ['Alice', 'Bob', '', 'David', 'Eve'],

'Age': [25, 30, np.nan, 35, 40],

'City': ['New York', 'Los Angeles', 'Chicago', '', 'Seattle'],

'Gender': ['F', 'M', 'M', 'M', 'F']

}

df = pd.DataFrame(data)

cleaned\_df = clean\_dataframe(df)

print("Cleaned DataFrame:")

print(cleaned\_df)

**2b.Write a function that detects and removes outliers from a numeric column in a**

**DataFrame using the IQR method.**

Ans:

import pandas as pd

def remove\_outliers(df, column):

Q1 = df[column].quantile(0.25)

Q3 = df[column].quantile(0.75)

IQR = Q3 - Q1

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

df\_cleaned = df[(df[column] >= lower\_bound) & (df[column] <= upper\_bound)]

return df\_cleaned

data = {

'Score': [85, 88, 84, 91, 70, 100, 99, 82, 81, 120, 79, 78]

}

df = pd.DataFrame(data)

cleaned\_df = remove\_outliers(df, 'Score')

print("Cleaned DataFrame after removing outliers:")

print(cleaned\_df)

**3. Lambda Functions and Map-Reduce:**

**3a. Use a lambda function to ϐilter out even numbers from a list of integers.**

Ans:

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

filtered = list(filter(lambda x: x % 2 != 0, numbers))

print("Original list:", numbers)

print("Filtered list (odd numbers only):", filtered)

**3b.Use the reduce function to calculate the product of the remaining numbers.**

Ans:

from functools import reduce

numbers = [1, 2, 3, 4, 5]

product = reduce(lambda x, y: x \* y, numbers)

print("List of numbers:", numbers)

print("Product of numbers:", product)

**3c.Use a lambda function with the ϐilter function to remove words from a list that**

**are shorter than 4 characters. Then, use reduce to concatenate the remaining**

**words into a single string.**

Ans:

from functools import reduce

words = ["mango", "banana", "pear", "kiwi", "orange"]

result\_string = reduce(lambda x, y: x + " " + y, filter(lambda word: len(word) >= 4, words))

print("List of words:", words)

print("Concatenated string:", result\_string)

data = {

'Date': ['2023-01-01','2023-01-02','2023-01-03', '2023-01-04','2023-01-05',

'2023-01-06','2023-01-07','2023-01-08','2023-01-09', '2023-01-10'] , 'Sales': [100, 120, 130, 150, 160, 180, 200, 210, 220, 230]

}df = pd.DataFrame(data)

#visualisation

plt.figure(figsize=(6, 6))

plt.plot(df['Date'], df['Sales'], marker='o', color='g', label='Sales Trend')

**4. Data Visualization:**

**4a.Using Matplotlib, create a line chart showing the trend of sales over time from**

**the given dataset.**

Ans:

data = {

'Date': ['2023-01-01','2023-01-02','2023-01-03', '2023-01-04','2023-01-05',

'2023-01-06','2023-01-07','2023-01-08','2023-01-09', '2023-01-10'] , 'Sales': [100, 120, 130, 150, 160, 180, 200, 210, 220, 230]

}df = pd.DataFrame(data)

plt.figure(figsize=(6, 6))

plt.plot(df['Date'], df['Sales'], marker='o', color='g', label='Sales Trend')



4b.Customize the chart with labels, title, and legend.

Ans:plt.title('Sales Trend Over Time')

plt.xlabel('Date')

plt.ylabel('Sales')

plt.legend()



**4c.Create a scatter plot showing the relationship between two numerical columns**

**in a DataFrame. Add a trend line to the scatter plot.**

Ans:

import numpy as np

np.random.seed(0)

x = np.random.randn(50)

y = 2 \* x + np.random.randn(50)

plt.figure(figsize=(8, 6))

plt.scatter(x, y, color='g', label='Data Points')

m, b = np.polyfit(x, y, 1) plt.plot(x, m \* x + b, color='r', label='Trend Line')



**5. Data Aggregation:**

**5a.Given a list of dictionaries representing transactions, write a function to**

**aggregate the total amount spent by each user.**

Ans:from collections import defaultdict

def aggregate\_total\_spent(transactions):

# Create a defaultdict to store total amount spent by each user

total\_spent = defaultdict(float)

for transaction in transactions:

user = transaction['user']

amount\_spent = transaction['amount']

total\_spent[user] += amount\_spent

return total\_spent

transactions = [

{'user': 'Alice', 'amount': 100.0},

{'user': 'Bob', 'amount': 50.0},

{'user': 'Alice', 'amount': 200.0},

{'user': 'Charlie', 'amount': 75.0},

{'user': 'Bob', 'amount': 150.0},

{'user': 'Alice', 'amount': 50.0},

]

total\_spent = aggregate\_total\_spent(transactions)

for user, total\_amount in total\_spent.items():

print(f"User: {user}, Total Amount Spent: {total\_amount}")

**5b.Write a function that calculates the moving average of the total amount spent by**

**each user over a speciϐied window size.**

Ans:from collections import defaultdict, deque

def moving\_average\_total\_spent(transactions, window\_size):

user\_total = defaultdict(list)

moving\_avg = defaultdict(float)

for transaction in transactions:

user = transaction['user']

amount = transaction['amount']

user\_total[user].append(amount)

if len(user\_total[user]) > window\_size:

user\_total[user].pop(0)

total\_spent = sum(user\_total[user])

moving\_avg[user] = total\_spent / len(user\_total[user])

return dict(moving\_avg)

window\_size = 3

moving\_avg = moving\_average\_total\_spent(transactions, window\_size)

print("\nMoving average of total amount spent by each user:")

print(moving\_avg)

**6. Exception Handling:**

**6a.Write a function that handles division by zero and returns a meaningful error**

**message when a division by zero occurs.**

Ans:def safe\_divide(a, b):

try:

result = a / b

except ZeroDivisionError:

return "Error: Division by zero is not allowed."

else:

return result

print(safe\_divide(10, 2))

print(safe\_divide(5, 0))

**6b.Write a function that takes a list of ϐile paths and attempts to open each one,**

**handling FileNotFoundError, PermissionError, and IOError, and logging the**

**results.**

Ans:

import logging

def open\_files(file\_paths):

logging.basicConfig(filename='file\_opening.log', level=logging.INFO, format='%(asctime)s - %(message)s', datefmt='%Y-%m-%d %H:%M:%S')

for file\_path in file\_paths:

try:

with open(file\_path, 'r') as file:

logging.info(f"Successfully opened file: {file\_path}")

except FileNotFoundError:

logging.error(f"FileNotFoundError: File not found: {file\_path}")

except PermissionError:

logging.error(f"PermissionError: Permission denied: {file\_path}")

except IOError:

logging.error(f"IOError: Unable to open file: {file\_path}")

except Exception as e:

logging.error(f"Unexpected error occurred while opening file {file\_path}: {str(e)}")

file\_paths = ['file1.txt', 'file2.txt', 'file3.txt', '/root/somefile.txt']

open\_files(file\_paths)

**7. Working with Dates:**

**7a.Write a function that takes a list of date strings in various formats and converts**

**them to a standardized format (YYYY-MM-DD).**

Ans:from datetime import datetime

def standardize\_date\_formats(date\_strings):

standardized\_dates = []

for date\_str in date\_strings:

try:

date\_obj = datetime.strptime(date\_str, '%Y-%m-%d')

except ValueError:

try:

date\_obj = datetime.strptime(date\_str, '%m/%d/%Y')

except ValueError:

try:

date\_obj = datetime.strptime(date\_str, '%d-%b-%Y')

except ValueError:

try:

date\_obj = datetime.strptime(date\_str, '%Y-%b-%d')

except ValueError:

print(f"Skipping invalid date format: {date\_str}")

continue

standardized\_dates.append(date\_obj.strftime('%Y-%m-%d'))

return standardized\_dates

**7b.Write a function that calculates the number of business days between two given**

**dates, excluding weekends and holidays.**

Ans:

from datetime import datetime, timedelta

def is\_business\_day(date):

return date.weekday() < 5

def calculate\_business\_days(start\_date, end\_date, holidays=[]):

current\_date = start\_date

business\_days\_count = 0

while current\_date <= end\_date:

if is\_business\_day(current\_date) and current\_date not in holidays:

business\_days\_count += 1

current\_date += timedelta(days=1)

return business\_days\_count

date\_strings = ['2023-12-25', '12/31/2024', '15-Aug-2025', '2026-Sep-30', 'InvalidDate']

standardized\_dates = standardize\_date\_formats(date\_strings)

print("Standardized Dates:")

print(standardized\_dates)

**8. ETL Process:**

**8a.Simulate an ETL process using Python that extracts data from a list of**

**dictionaries, transforms it by normalizing numeric ϐields, and loads it into a**

**Pandas DataFrame.**

Ans:

import pandas as pd

data = [

{'id': 1, 'name': 'Alice', 'age': 25, 'salary': 5000},

{'id': 2, 'name': 'Bob', 'age': 30, 'salary': 7500},

{'id': 3, 'name': 'Charlie', 'age': 28, 'salary': 9000},

{'id': 4, 'name': 'David', 'age': 35, 'salary': 6000},

{'id': 5, 'name': 'Eve', 'age': 27, 'salary': 8000}

]

for entry in data:

entry['normalized\_salary'] = entry['salary'] / 100 # Transforming by normalizing salary (dividing by 100)

df = pd.DataFrame(data)

print("Transformed Data:")

print(df)

**8b.Extend the ETL process to include a validation step that checks for data quality**

**issues (e.g., missing values, outliers) before loading the data into the DataFrame.**

Ans:

import pandas as pd

data = [

{'id': 1, 'name': 'Alice', 'age': 25, 'salary': 5000},

{'id': 2, 'name': 'Bob', 'age': 30, 'salary': 7500},

{'id': 3, 'name': 'Charlie', 'age': None, 'salary': 9000}, # Simulate missing age

{'id': 4, 'name': 'David', 'age': 35, 'salary': 6000},

{'id': 5, 'name': 'Eve', 'age': 27, 'salary': 8000}

]

for entry in data:

if entry['age'] is None:

entry['age'] = 0

entry['normalized\_salary'] = entry['salary'] / 100

df = pd.DataFrame(data)

def validate\_data\_quality(df):

issues\_found = False

for column in df.columns:

if df[column].isnull().any():

issues\_found = True

print(f"Warning: Missing values found in column '{column}'")

if column == 'salary':

salary\_std = df['salary'].std()

salary\_mean = df['salary'].mean()

salary\_threshold = salary\_mean + 2 \* salary\_std

if (df['salary'] > salary\_threshold).any():

issues\_found = True

print("Warning: Outliers found in 'salary' column")

if not issues\_found:

print("No data quality issues found. Data is clean.")

print("Data Validation:")

validate\_data\_quality(df)

print("\nTransformed Data:")

print(df)

**9. Data Normalization:**

**9a.Write a function that normalizes the values in a DataFrame column to a range**

**between 0 and 1.**

Ans:import pandas as pd

def normalize\_column(df, column\_name):

column = df[column\_name]

minVal = column.min()

maxVal = column.max()

normalized\_values = (column - minVal) / (maxVal - minVal)

df[column\_name] = normalized\_values

return df

data = {

'A': [1, 2, 3, 4, 5],

'B': [10, 20, 30, 40, 50],

'C': [100, 200, 300, 400, 500]

}

df = pd.DataFrame(data)

normalized\_df = normalize\_column(df.copy(), 'B')

print("Normalized DataFrame (column B):")

print(normalized\_df)

**9b.Write a function that standardizes the values in a DataFrame column (mean=0,**

**standard deviation=1).**

Ans: import pandas as pd

def standardize\_column(df, column\_name):

column = df[column\_name]

mean\_val = column.mean()

stdVal = column.std()

standardized\_values = (column - mean\_val) / stdVal

df[column\_name] = standardized\_values

return df

data = {

'A': [1, 2, 3, 4, 5],

'B': [10, 20, 30, 40, 50],

'C': [100, 200, 300, 400, 500]

}

df = pd.DataFrame(data)

standardized\_df = standardize\_column(df.copy(), 'C')

print("\nStandardized DataFrame (column C):")

print(standardized\_df)

**10. Advanced List Comprehensions:**

**10a.Given a list of numbers, create a new list containing the square roots of the even**

**numbers only, using list comprehension.**

Ans:

import math

def square\_roots\_of\_even\_numbers(numbers):

square\_roots = [math.sqrt(num) for num in numbers if num % 2 == 0]

return square\_roots

numbers = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

square\_roots\_even = square\_roots\_of\_even\_numbers(numbers)

print("Square roots of even numbers:", square\_roots\_even)

**10b.Given a list of tuples representing (name, score), create a new list containing**

**names of students who scored above the average, using list comprehension.**

Ans:

def names\_above\_average(scores):

if not scores:

return []

total\_score = sum(score for \_, score in scores)

average\_score = total\_score / len(scores)

above\_average\_names = [name for name, score in scores if score > average\_score]

return above\_average\_names

student\_scores = [("Alice", 80), ("Bob", 75), ("Charlie", 90), ("David", 85), ("Eve", 95)]

above\_average\_students = names\_above\_average(student\_scores)

print("Students who scored above the average:", above\_average\_students)

**11. Unit Testing:**

**11.a Write unit tests for a function that calculates the factorial of a number. Use the**

**unittest framework.**

Ans:

import unittest

def factorial(n):

if n < 0:

raise ValueError("Factorial is not defined for negative numbers")

if n == 0:

return 1

result = 1

for i in range(1, n + 1):

result \*= i

return result

class TestFactorialFunction(unittest.TestCase):

def test\_factorial\_zero(self):

self.assertEqual(factorial(0), 1)

def test\_factorial\_positive(self):

self.assertEqual(factorial(5), 120)

def test\_factorial\_negative(self):

with self.assertRaises(ValueError):

factorial(-1)

def test\_factorial\_large\_number(self):

self.assertEqual(factorial(10), 3628800)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**11.b Write unit tests for a function that checks if a given string is a palindrome.**

Ans:

import unittest

def is\_palindrome(s):

s = s.lower().replace(" ", "")

return s == s[::-1]

class TestIsPalindromeFunction(unittest.TestCase):

def test\_palindrome\_true(self):

self.assertTrue(is\_palindrome("radar"))

self.assertTrue(is\_palindrome("A Santa at NASA"))

def test\_palindrome\_false(self):

self.assertFalse(is\_palindrome("hello"))

self.assertFalse(is\_palindrome("not a palindrome"))

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**12. Decorators:**

**12. a Create a decorator that logs the execution time of a function. Apply it to a**

**function that sorts a large list.**

Ans:

import time

from functools import wraps

def log\_execution\_time(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

result = func(\*args, \*\*kwargs)

end\_time = time.time()

execution\_time = end\_time - start\_time

print(f"Execution time of {func.\_\_name\_\_}: {execution\_time:.4f} seconds")

return result

return wrapper

@log\_execution\_time

def sort\_large\_list(large\_list):

return sorted(large\_list)

if \_\_name\_\_ == '\_\_main\_\_':

large\_list = list(range(1000, 0, -1))

sorted\_list = sort\_large\_list(large\_list)

print(f"Sorted list: {sorted\_list[:10]}... (truncated)")

**12.b Create a decorator that retries a function up to 3 times if it raises an exception,**

**with a delay between retries**

Ans:

import time

from functools import wraps

def retry(max\_retries=3, delay=1):

def decorator\_retry(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

retries = 0

while retries < max\_retries:

try:

return func(\*args, \*\*kwargs)

except Exception as e:

print(f"Attempt {retries + 1} failed: {str(e)}")

retries += 1

time.sleep(delay)

raise RuntimeError(f"Function {func.\_\_name\_\_} failed after {max\_retries} retries")

return wrapper

return decorator\_retry

@retry(max\_retries=3, delay=2)

def unreliable\_function():

import random

if random.random() < 0.7:

raise ValueError("Random failure")

return "Success"

if \_\_name\_\_ == '\_\_main\_\_':

try:

result = unreliable\_function()

print(f"Function returned: {result}")

except RuntimeError as e:

print(f"Function failed: {str(e)}")

**13. Concurrency with Threads:**

**13.a Write a program that uses threading to calculate the sum of a large list of**

**numbers by dividing the work among multiple threads.**

Ans:

import threading

def sum\_sublist(sublist, result):

result\_sum = sum(sublist)

result.append(result\_sum)

def calculate\_sum\_with\_threads(numbers, num\_threads):

chunk\_size = (len(numbers) + num\_threads - 1) // num\_threads

chunks = [numbers[i:i + chunk\_size] for i in range(0, len(numbers), chunk\_size)]

result = []

threads = []

for chunk in chunks:

thread = threading.Thread(target=sum\_sublist, args=(chunk, result))

thread.start()

threads.append(thread)

for thread in threads:

thread.join()

return sum(result)

if \_\_name\_\_ == '\_\_main\_\_':

large\_list = list(range(1, 1000001))

num\_threads = 4

total\_sum = calculate\_sum\_with\_threads(large\_list, num\_threads)

print(f"Total sum using {num\_threads} threads: {total\_sum}")

**13.b Write a program that uses threading to fetch data from multiple URLs concurrently and print the status code of each response.**

Ans:

import threading

import requests

def fetch\_url(url):

try:

response = requests.get(url)

print(f"URL: {url}, Status Code: {response.status\_code}")

except requests.RequestException as e:

print(f"URL: {url}, Exception: {str(e)}")

def fetch\_urls\_with\_threads(urls):

# Create threads for each URL

threads = []

for url in urls:

thread = threading.Thread(target=fetch\_url, args=(url,))

thread.start()

threads.append(thread)

for thread in threads:

thread.join()

if \_\_name\_\_ == '\_\_main\_\_':

urls = [

"<https://jsonplaceholder.typicode.com/posts/1>",

"<https://jsonplaceholder.typicode.com/posts/2>",

"<https://jsonplaceholder.typicode.com/posts/3>",

"<https://jsonplaceholder.typicode.com/posts/4>",

"<https://jsonplaceholder.typicode.com/posts/5>"

]

fetch\_urls\_with\_threads(urls)

**14. Data Pipeline Simulation:**

**14.a Simulate a data pipeline that processes a list of dictionaries, applying various**

**transformations, and outputs the processed data as a list of dictionaries.**

Ans:

data = [

{"id": 1, "name": "Akash", "age": 28, "city": "Mumbai"},

{"id": 2, "name": "Aarti", "age": 32, "city": "Delhi"},

{"id": 3, "name": "Arjun", "age": 30, "city": "Bangalore"},

{"id": 4, "name": "Priya", "age": 25, "city": "Chennai"},

{"id": 5, "name": "Ravi", "age": 35, "city": "Hyderabad"}

]

def filter\_records(records, min\_age, max\_age):

filtered\_data = [record for record in records if min\_age <= record['age'] <= max\_age]

return filtered\_data

def map\_records(records, fields):

mapped\_data = [{field: record.get(field, None) for field in fields} for record in records]

return mapped\_data

def data\_pipeline(data):

filtered\_data = filter\_records(data, 25, 30)

mapped\_data = map\_records(filtered\_data, ['name', 'age'])

return mapped\_data

if \_\_name\_\_ == '\_\_main\_\_':

processed\_data = data\_pipeline(data)

print("Processed Data:")

for record in processed\_data:

print(record)

**14.b Extend the pipeline to include an error-handling stage that logs any errors**

**encountered during processing.**

Ans:

data = [

{"id": 1, "name": "Akash", "age": 28, "city": "Mumbai"},

{"id": 2, "name": "Aarti", "age": 32, "city": "Delhi"},

{"id": 3, "name": "Arjun", "age": 30, "city": "Bangalore"},

{"id": 4, "name": "Priya", "age": 25, "city": "Chennai"},

{"id": 5, "name": "Ravi", "age": 35, "city": "Hyderabad"},

{"id": 6, "name": "Neha", "age": "unknown", "city": "Kolkata"} # Intentional error for age

]

def filter\_records(records, min\_age, max\_age):

filtered\_data = []

for record in records:

try:

age = int(record['age'])

if min\_age <= age <= max\_age:

filtered\_data.append(record)

except ValueError as e:

print(f"Error filtering record {record}: {str(e)}")

return filtered\_data

def map\_records(records, fields):

mapped\_data = []

for record in records:

mapped\_record = {}

for field in fields:

mapped\_record[field] = record.get(field, None)

mapped\_data.append(mapped\_record)

return mapped\_data

def data\_pipeline(data):

try:

filtered\_data = filter\_records(data, 25, 30)

mapped\_data = map\_records(filtered\_data, ['name', 'age'])

return mapped\_data

except Exception as e:

print(f"Error in data pipeline: {str(e)}")

return []

if \_\_name\_\_ == '\_\_main\_\_':

processed\_data = data\_pipeline(data)

print("\nProcessed Data:")

for record in processed\_data:

print(record)

**15. Configuration Management:**

**15.a Write a Python script that reads configuration settings from a dictionary and**

**uses them to perform a specific task.**

Ans:

config = {

"username": "admin",

"password": "secretpassword",

"host": "localhost",

"port": 5432,

"database": "mydatabase"

}

def perform\_task\_with\_config(config):

print(f"Connecting to database {config['database']} on {config['host']}:{config['port']}...")

if \_\_name\_\_ == "\_\_main\_\_":

perform\_task\_with\_config(config)

**15.b Write a function that validates the configuration settings, ensuring that all**

**required fields are present and have valid values.**

Ans:

def validate\_config(config):

required\_fields = ["username", "password", "host", "port", "database"]

for field in required\_fields:

if field not in config:

return False, f"Missing required field: {field}"

elif not config[field]:

return False, f"Empty value for field: {field}"

return True, "Configuration is valid"

if \_\_name\_\_ == "\_\_main\_\_":

# Validate configuration settings

is\_valid, message = validate\_config(config)

if is\_valid:

print("Configuration is valid")

else:

print(f"Invalid configuration: {message}")

**16. Handling Large Data Sets:**

**16.a Write a function that processes a large list of numbers in chunks and calculates**

**the average value of the list.**

Ans:

def process\_large\_numbers(numbers, chunk\_size):

total\_sum = 0

count = 0

for i in range(0, len(numbers), chunk\_size):

chunk = numbers[i:i + chunk\_size]

total\_sum += sum(chunk)

count += len(chunk)

return total\_sum / count if count > 0 else 0

if \_\_name\_\_ == "\_\_main\_\_":

numbers = list(range(1, 1000001))

chunk\_size = 10000

average = process\_large\_numbers(numbers, chunk\_size)

print(f"Average value of the list: {average}")

**16.b Write a function that processes a large list of strings in chunks, counts the**

**frequency of each string, and returns a dictionary with the results.**

Ans:

from collections import defaultdict

def process\_large\_strings(strings, chunk\_size):

frequency\_dict = defaultdict(int)

for i in range(0, len(strings), chunk\_size):

chunk = strings[i:i + chunk\_size]

for s in chunk:

frequency\_dict[s] += 1

return dict(frequency\_dict)

if \_\_name\_\_ == "\_\_main\_\_":

strings = ["apple", "banana", "apple", "orange", "banana", "apple", "apple", "banana"] \* 100000

chunk\_size = 10000

frequency = process\_large\_strings(strings, chunk\_size)

print(f"Frequency dictionary: {frequency}")

**17. Class and Objects:**

**17.a Create a class representing a bank account with methods to deposit, withdraw,**

**and check balance. Ensure proper error handling for invalid operations**

Ans:

class BankAccount:

def \_\_init\_\_(self, account\_number, initial\_balance=0):

self.account\_number = account\_number

self.balance = initial\_balance

def deposit(self, amount):

if amount > 0:

self.balance += amount

print(f"Deposit of ${amount:.2f} successful.")

else:

print("Invalid deposit amount. Please enter a positive number.")

def withdraw(self, amount):

if amount > 0:

if self.balance >= amount:

self.balance -= amount

print(f"Withdrawal of ${amount:.2f} successful.")

else:

print("Insufficient balance. Withdrawal not processed.")

else:

print("Invalid withdrawal amount. Please enter a positive number.")

def check\_balance(self):

return self.balance

account = BankAccount('1234567890', 1000)

account.deposit(500)

print(f"Current balance: ${account.check\_balance():.2f}")

account.withdraw(200)

print(f"Current balance: ${account.check\_balance():.2f}")

account.withdraw(-100)

account.deposit(-50)

**17.b Extend the bank account class to support multiple currencies, with methods to**

**convert between currencies using a given exchange rate.**

Ans:

class BankAccount:

def \_\_init\_\_(self, account\_number, balance=0, currency="INR"):

self.account\_number = account\_number

self.balance = balance

self.currency = currency

def deposit(self, amount):

if amount > 0:

self.balance += amount

print(f"Deposited {amount} {self.currency}. New balance is {self.balance} {self.currency}.")

else:

print("Deposit amount must be greater than zero.")

def withdraw(self, amount):

if 0 < amount <= self.balance:

self.balance -= amount

print(f"Withdrew {amount} {self.currency}. New balance is {self.balance} {self.currency}.")

else:

print("Invalid amount or insufficient funds.")

def check\_balance(self):

print(f"Current balance of account {self.account\_number} is {self.balance} {self.currency}.")

def set\_currency(self, currency):

self.currency = currency

def convert\_to(self, new\_currency, exchange\_rate):

if new\_currency == self.currency:

print("Cannot convert to the same currency.")

return

if exchange\_rate > 0:

new\_balance = self.balance \* exchange\_rate

print(f"Converted {self.balance} {self.currency} to {new\_balance} {new\_currency} at rate {exchange\_rate}.")

self.balance = new\_balance

self.currency = new\_currency

else:

print("Invalid exchange rate.")

account1 = BankAccount("123456", 4000, "INR")

account1.check\_balance()

account1.deposit(1000)

account1.check\_balance()

account1.withdraw(2500)

account1.check\_balance()

account1.convert\_to("USD", 0.014)

account1.check\_balance()

**18. Regular Expressions:**

**18.a Write a function that validates email addresses using regular expressions.**

Ans:

import re

def validate\_email(email):

pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'

regex = re.compile(pattern)

if regex.match(email):

return True

else:

return False

email1 = "arun123@example.com"

email2 = "invalid\_email@"

print(f"{email1} is valid: {validate\_email(email1)}")

print(f"{email2} is valid: {validate\_email(email2)}")

**18.b Write a function that extracts all the dates from a given text string in the format**

**(DD-MM-YYYY).**

Ans:

import re

def extract\_dates(text):

date\_pattern = r'\b(\d{2}-\d{2}-\d{4})\b'

dates = re.findall(date\_pattern, text)

return dates

texts = "The project deadline is on 29-07-2024. Please submit the report by then. Another important date is 11-08-2024."

extracted\_dates = extract\_dates(texts)

print("Extracted dates:", extracted\_dates)

**19. Data Encryption:**

**19.a Write a Python script that encrypts and decrypts text using the Fernet symmetric**

**encryption from the cryptography library.**

Ans:

pip install cryptography

from cryptography.fernet import Fernet

def generate\_key():

return Fernet.generate\_key()

def encrypt\_message(message, key):

fernet = Fernet(key)

encrypted\_message = fernet.encrypt(message.encode())

return encrypted\_message

def decrypt\_message(encrypted\_message, key):

fernet = Fernet(key)

decrypted\_message = fernet.decrypt(encrypted\_message).decode()

return decrypted\_message

if \_\_name\_\_ == "\_\_main\_\_":

key = generate\_key()

print(f"Generated Key: {key.decode()}")

plaintext\_message = "Hello, this is a secret message!"

encrypted\_message = encrypt\_message(plaintext\_message, key)

print(f"Encrypted Message: {encrypted\_message}")

decrypted\_message = decrypt\_message(encrypted\_message, key)

print(f"Decrypted Message: {decrypted\_message}")

**19.b Write a function that encrypts and decrypts a dictionary of sensitive data,**

**preserving the structure of the dictionary**

Ans:

pip install cryptography

from cryptography.fernet import Fernet

import json

def generate\_key():

return Fernet.generate\_key()

def encrypt\_dict(data\_dict, key):

json\_data = json.dumps(data\_dict).encode()

fernet = Fernet(key)

encrypted\_data = fernet.encrypt(json\_data)

return encrypted\_data

def decrypt\_dict(encrypted\_data, key):

fernet = Fernet(key)

decrypted\_data = fernet.decrypt(encrypted\_data)

decrypted\_json\_data = decrypted\_data.decode()

data\_dict = json.loads(decrypted\_json\_data)

return data\_dict

if \_\_name\_\_ == "\_\_main\_\_":

key = generate\_key()

print(f"Generated Key: {key.decode()}")

sensitive\_data = {

"name": "Arjun",

"age": 35,

"address": {

"street": "13 Silk Board",

"city": "Bangalore"

}

}

encrypted\_data = encrypt\_dict(sensitive\_data, key)

print(f"Encrypted Data: {encrypted\_data}")

decrypted\_data = decrypt\_dict(encrypted\_data, key)

print(f"Decrypted Data: {decrypted\_data}")

**20. Memory Management:**

**20.a Write a program to monitor memory usage of a Python script and log it to the**

**console at regular intervals.**

Ans:

import psutil

import time

def monitor\_memory(interval\_seconds=1, duration\_seconds=10):

print(f"Monitoring memory usage every {interval\_seconds} seconds for {duration\_seconds} seconds...")

start\_time = time.time()

while (time.time() - start\_time) < duration\_seconds:

memory\_usage = psutil.virtual\_memory()

print(f"Memory Usage - Total: {memory\_usage.total / (1024\*1024):.2f} MB, "

f"Available: {memory\_usage.available / (1024\*1024):.2f} MB, "

f"Used: {memory\_usage.used / (1024\*1024):.2f} MB, "

f"Free: {memory\_usage.free / (1024\*1024):.2f} MB, "

f"Percentage: {memory\_usage.percent:.2f}%")

time.sleep(interval\_seconds)

if \_\_name\_\_ == "\_\_main\_\_":

monitor\_memory(interval\_seconds=2, duration\_seconds=30)

**20.b Write a function that generates a large list of random numbers and uses memory**

**profiling to identify any memory leaks.**

Ans:

pip install memory\_profiler

from memory\_profiler import profile

import random

@profile

def generate\_large\_list(size):

large\_list = [random.random() for \_ in range(size)]

return large\_list

if \_\_name\_\_ == "\_\_main\_\_":

generate\_large\_list(1000000)

**21. Parallel Processing:**

**21.a Use the multiprocessing module to parallelize a CPU-bound task, such as**

**calculating the prime numbers in a given range.**

Ans:

import multiprocessing

def is\_prime(n):

if n <= 1:

return False

if n <= 3:

return True

if n % 2 == 0 or n % 3 == 0:

return False

i = 5

while i \* i <= n:

if n % i == 0 or n % (i + 2) == 0:

return False

i += 6

return True

def calculate\_primes(start, end):

primes = []

for number in range(start, end + 1):

if is\_prime(number):

primes.append(number)

return primes

def parallel\_prime\_calculation(num\_processes, start, end):

chunk\_size = (end - start + 1) // num\_processes

pool = multiprocessing.Pool(processes=num\_processes)

results = []

for i in range(num\_processes):

chunk\_start = start + i \* chunk\_size

chunk\_end = chunk\_start + chunk\_size - 1 if i < num\_processes - 1 else end

results.append(pool.apply\_async(calculate\_primes, (chunk\_start, chunk\_end)))

pool.close()

pool.join()

primes = []

for result in results:

primes.extend(result.get())

return primes

if \_\_name\_\_ == "\_\_main\_\_":

start\_range = 1

end\_range = 10000

num\_processes = 4

print(f"Calculating prime numbers from {start\_range} to {end\_range} using {num\_processes} processes...")

primes = parallel\_prime\_calculation(num\_processes, start\_range, end\_range)

print(f"Found {len(primes)} prime numbers.")

print(primes)

**21.b Use the multiprocessing module to parallelize a CPU-bound task, such as**

**calculating the prime numbers in a given range.**

Ans:

import numpy as np

import multiprocessing

def matrix\_multiplication\_chunk(matrix1, matrix2\_chunk):

return np.dot(matrix1, matrix2\_chunk)

def parallel\_matrix\_multiplication(matrix1, matrix2, num\_processes):

if matrix1.shape[1] != matrix2.shape[0]:

raise ValueError("Matrix dimensions are not compatible for multiplication.")

chunk\_size = matrix2.shape[1] // num\_processes

pool = multiprocessing.Pool(processes=num\_processes)

results = []

for i in range(num\_processes):

chunk\_start = i \* chunk\_size

chunk\_end = chunk\_start + chunk\_size

chunk\_matrix2 = matrix2[:, chunk\_start:chunk\_end]

results.append(pool.apply\_async(matrix\_multiplication\_chunk, (matrix1, chunk\_matrix2)))

pool.close()

pool.join()

result\_matrix = np.hstack([res.get() for res in results])

return result\_matrix

if \_\_name\_\_ == "\_\_main\_\_":

matrix1 = np.random.rand(2, 4)

matrix2 = np.random.rand(4, 5)

num\_processes = 3

print("Matrix 1:")

print(matrix1)

print("Matrix 2:")

print(matrix2)

result\_matrix = parallel\_matrix\_multiplication(matrix1, matrix2, num\_processes)

print("Result Matrix (Parallel Multiplication):")

print(result\_matrix)

**22. Error Handling:**

**22.a Write a function that raises and handles custom exceptions to manage specific**

**error cases in a given task.**

Ans:

class FactorialError(Exception):

def \_\_init\_\_(self, message):

self.message = message

super().\_\_init\_\_(self.message)

def calculate\_factorial(n):

if not isinstance(n, int):

raise FactorialError("Factorial can only be calculated for integers")

if n < 0:

raise FactorialError("Factorial is not defined for negative numbers")

if n == 0:

return 1

factorial = 1

for i in range(1, n + 1):

factorial \*= i

return factorial

inputs = [5, 0, -1, 10.5, 'abc']

for num in inputs:

try:

result = calculate\_factorial(num)

print(f"The factorial of {num} is: {result}")

except ValueError:

print(f"Error: Input must be a valid integer")

except FactorialError as e:

print(f"FactorialError: {e.message}")

except Exception as e:

print(f"Unexpected Error: {str(e)}")

**22.b Write a function that uses context managers to handle resources (e.g., database**

**connections) and properly handles exceptions that may occur during resource**

**usage.**

Ans:

import sqlite3

def perform\_database\_operations(database\_file):

try:

with sqlite3.connect(database\_file) as conn:

cursor = conn.cursor()

cursor.execute('''CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY,

name TEXT NOT NULL,

age INTEGER

)''')

cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ('Arun', 32))

conn.commit()

cursor.execute("SELECT \* FROM users")

rows = cursor.fetchall()

print("Retrieved rows:")

for row in rows:

print(row)

except sqlite3.Error as e:

print(f"Database error: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

database\_file = "example.db"

perform\_database\_operations(database\_file)

**23. Recursion:**

**23.a. Write a recursive function to calculate the nth Fibonacci number**

Ans:

def fibonacci(n):

if n <= 0:

return 0

elif n == 1:

return 1

else:

return fibonacci(n-1) + fibonacci(n-2)

n = 9

result = fibonacci(n)

print(f"The {n}th Fibonacci number is: {result}")

**23.b. Write a recursive function to solve the Tower of Hanoi problem.**

Ans:

def tower\_of\_hanoi(n, source, target, temp):

if n == 1:

print(f"Move disk 1 from {source} to {target}")

return

else:

tower\_of\_hanoi(n-1, source, temp, target)

print(f"Move disk {n} from {source} to {target}")

tower\_of\_hanoi(n-1, temp, target, source)

n = 4

tower\_of\_hanoi(n, 'A', 'C', 'B')

**24. Data Merging:**

**24.a Given two lists of dictionaries, write a function to merge them based on a**

**common key.**

Ans:

def merge\_lists\_of\_dicts(list1, list2, common\_key):

merged\_list = []

dict2 = {item[common\_key]: item for item in list2}

for item1 in list1:

key = item1[common\_key]

if key in dict2:

merged\_item = {\*\*item1, \*\*dict2[key]} # Merge dictionaries

merged\_list.append(merged\_item)

else:

merged\_list.append(item1) # If no match found, add item from list1

for item2 in list2:

key = item2[common\_key]

if key not in dict2:

merged\_list.append(item2)

return merged\_list

list1 = [

{'id': 1, 'name': 'Alice', 'age': 25},

{'id': 2, 'name': 'Bob', 'age': 30},

{'id': 3, 'name': 'Charlie', 'age': 35}

]

list2 = [

{'id': 2, 'city': 'New York'},

{'id': 3, 'city': 'Los Angeles'},

{'id': 4, 'city': 'Chicago'}

]

merged\_list = merge\_lists\_of\_dicts(list1, list2, 'id')

print("Merged List:")

print(merged\_list)

**24.b Write a function that merges multiple DataFrames based on a common key and**

**handles conflicts by keeping the most recent data.**

Ans:

import pandas as pd

def merge\_dataframes\_on\_key(dataframes, key):

merged\_df = dataframes[0]

for df in dataframes[1:]:

merged\_df = pd.merge(merged\_df, df, on=key, how='outer')

merged\_df = merged\_df.sort\_values(by=key)

merged\_df = merged\_df.drop\_duplicates(subset=key, keep='last')

return merged\_df

df1 = pd.DataFrame({'id': [1, 2, 3], 'value': ['A', 'B', 'C']})

df2 = pd.DataFrame({'id': [2, 3, 4], 'value': ['D', 'E', 'F']})

df3 = pd.DataFrame({'id': [1, 3, 4], 'value': ['G', 'H', 'I']})

dataframes = [df1, df2, df3]

merged\_df = merge\_dataframes\_on\_key(dataframes, 'id')

print("Merged DataFrame:")

print(merged\_df)

**25. Statistical Analysis:**

**25.a Write a function that calculates the mean, median, and mode of a list of numbers.**

Ans:

import numpy as np

from scipy import stats

def calculate\_statistics(numbers):

if not numbers:

raise ValueError("List of numbers is empty")

mean = np.mean(numbers)

median = np.median(numbers)

mode\_result = stats.mode(numbers)

mode = mode\_result.mode.tolist()

return mean, median, mode

numbers = [1, 2, 3, 4, 5, 5, 6, 6, 7, 8, 9]

try:

mean, median, mode = calculate\_statistics(numbers)

print(f"List of Numbers: {numbers}")

print(f"Mean: {mean}")

print(f"Median: {median}")

print(f"Mode: {mode}")

except ValueError as e:

print(f"Error: {e}")

**25.b Write a function that calculates the standard deviation and variance of a list of**

**numbers.**

Ans:

import numpy as np

def calculate\_std\_and\_var(numbers):

if not numbers:

raise ValueError("List of numbers is empty")

variance = np.var(numbers)

std\_deviation = np.std(numbers)

return std\_deviation, variance

numbers = [1, 2, 3, 4, 5, 5, 6, 6, 7, 8, 9]

try:

std\_deviation, variance = calculate\_std\_and\_var(numbers)

print(f"List of Numbers: {numbers}")

print(f"Standard Deviation: {std\_deviation:.2f}")

print(f"Variance: {variance:.2f}")

except ValueError as e:

print(f"Error: {e}")