**Python Assignment**

**1. Data Manipulation with Pandas:**

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

**the following tasks:**

§ Remove rows with missing values.

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

§ Sort the results by total sales in descending order.

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

**by month.**

import pandas as pd

data = {

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

'Month': [1, 1, 2, 2, 1, 2],

'Sales': [100, 200, 150, 300, 180, 210]

}

df = pd.DataFrame(data)

df.dropna(inplace=True)

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

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

print("Total Sales by Product:")

print(product\_sales)

**# Create pivot table showing sum of sales by product and month**

pivot\_table = pd.pivot\_table(df, values='Sales', index='Product', columns='Month', aggfunc='sum', fill\_value=0)

print("\nPivot Table - Sales by Product and Month:")

print(pivot\_table)

**2. Data Cleaning:**

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

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

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

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

**§ Convert all text to lowercase.import pandas as pd**

import numpy as np

data = {

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

'Month': [1, 1, 2, np.nan, 1, 2],

'Sales': [100, 200, np.nan, 300, 180, 210]

}

df = pd.DataFrame(data)

def clean\_dataframe(df):

# Replace empty strings with NaN

df.replace('', pd.NA, inplace=True)

numeric\_cols = df.select\_dtypes(include='number').columns

df[numeric\_cols] = df[numeric\_cols].fillna(df[numeric\_cols].mean())

text\_cols = df.select\_dtypes(include='object').columns

df[text\_cols] = df[text\_cols].apply(lambda x: x.str.lower())

return df

cleaned\_df = clean\_dataframe(df)

print(cleaned\_df)

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

**DataFrame using the IQR method.**

import pandas as pd

data = {'value': [1, 2, 3, 4, 5, 6, 7, 8, 9, 100]}

df = pd.DataFrame(data)

def remove\_outliers\_iqr(df, column):

Q1 = df[column].quantile(0.25)

Q3 = df[column].quantile(0.75)

IQR = Q3 - Q1

lower\_cutoff = Q1 - 1.5 \* IQR

upper\_cutoff = Q3 + 1.5 \* IQR

df\_filtered = df[(df[column] >= lower\_cutoff) & (df[column] <= upper\_cutoff)]

return df\_filtered

df\_filtered = remove\_outliers\_iqr(df, 'value')

print("Original DataFrame:")

print(df)

print("\nDataFrame with outliers removed:")

print(df\_filtered)

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

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

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

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

print("Original list of numbers:", numbers)

print("Filtered even numbers:", filtered\_numbers)

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

from functools import reduce

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

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

def filter\_not\_in\_filtered\_numbers(acc, x):

if x not in filtered\_numbers:

return acc \* x

else:

return acc

product = reduce(filter\_not\_in\_filtered\_numbers, numbers, 1)

**Use a lambda function with the filter 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.**

from functools import reduce

words = ['apple', 'cat', 'banana', 'dog', 'elephant']

filtered\_words = list(filter(lambda x: len(x) >= 4, words))

concatenated\_string = reduce(lambda x, y: x + ' ' + y, filtered\_words)

print("Original list of words:", words)

print("Filtered words longer than 4 characters:", filtered\_words)

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

**4. Data Visualization:**

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

**the given dataset.**

**Customize the chart with labels, title, and legend.**

import matplotlib.pyplot as plt

time\_periods = [1, 2, 3, 4, 5]

sales\_data = [50, 55, 70, 65, 80]

plt.figure(figsize=(10, 6)) # Optional: adjust figure size

plt.plot(time\_periods, sales\_data, marker='o', linestyle='-', color='b', label='Sales Trend')

plt.xlabel('Time Period')

plt.ylabel('Sales')

plt.title('Sales Trend Over Time')

plt.legend()

plt.grid(True) # Optional: add grid

plt.tight\_layout() # Optional: improve spacing

plt.show()

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

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

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

data = {

'X': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],

'Y': [2, 3.5, 4, 6.5, 6, 8, 9, 10, 12, 12.5]

}

df = pd.DataFrame(data)

X = df['X']

Y = df['Y']

plt.figure(figsize=(8, 6)) # Optional: adjust figure size

plt.scatter(X, Y, color='b', label='Data Points')

m, b = np.polyfit(X, Y, 1) # Fit a linear trend line

plt.plot(X, m\*X + b, color='r', label='Trend Line')

plt.xlabel('X')

plt.ylabel('Y')

plt.title('Scatter Plot with Trend Line')

plt.legend()

plt.grid(True) # Optional: add grid

plt.tight\_layout() # Optional: improve spacing

plt.show()

**5. Data Aggregation:**

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

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

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

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

from collections import defaultdict, deque

def aggregate\_total\_spent(transactions):

total\_spent = defaultdict(float)

for transaction in transactions:

user = transaction['user']

amount = transaction['amount']

total\_spent[user] += amount

return dict(total\_spent)

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

user\_amounts = defaultdict(deque)

moving\_averages = defaultdict(list)

for transaction in transactions:

user = transaction['user']

amount = transaction['amount']

user\_amounts[user].append(amount)

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

user\_amounts[user].popleft()

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

moving\_average = sum(user\_amounts[user]) / window\_size

moving\_averages[user].append(moving\_average)

return dict(moving\_averages)

transactions = [

{'user': 'A', 'amount': 100},

{'user': 'B', 'amount': 150},

{'user': 'A', 'amount': 75},

{'user': 'C', 'amount': 200},

{'user': 'B', 'amount': 50},

{'user': 'A', 'amount': 125},

{'user': 'C', 'amount': 300},

]

# Test aggregate\_total\_spent function

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

print("Total amount spent by each user:", total\_spent)

# Test moving\_average\_total\_spent function

window\_size = 2

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

print(f"Moving average of total amount spent by each user (window size {window\_size}):", moving\_averages)

#output  
Total amount spent by each user: {'A': 300.0, 'B': 200.0, 'C': 500.0}

Moving average of total amount spent by each user (window size 2): {'A': [87.5, 100.0], 'B': [100.0], 'C': [250.0]}

**6. Exception Handling:**

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

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

def safe\_divide(a,b

try:

result = a/b

return result

except ZeroDivisionError:

raise ZeroDivisionError("Division by zero is not allowed.")

**Write a function that takes a list of file paths and attempts to open each one,**

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

**results.**

import logging

def open\_files(file\_paths):

logging.basicConfig(filename='file\_handling.log', level=logging.INFO, format='%(asctime)s - %(message)s')

for file\_path in file\_paths:

try:

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

lines = file.readlines()

logging.info(f"File '{file\_path}' opened successfully. Read {len(lines)} lines.")

except FileNotFoundError:

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

except PermissionError:

logging.error(f"Permission denied to open file '{file\_path}'.")

except IOError as e:

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

except Exception as e:

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

**7. Working with Dates:**

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

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

from datetime import datetime

def standardize\_dates(date\_list):

standardized\_dates = []

for date\_str in date\_list:

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:

print(f"Error: Date '{date\_str}' format not recognized.")

continue

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

return standardized\_dates

dates = ["2023-01-15", "12/25/2023", "01-Mar-2023", "2023-04-30", "Invalid Date"]

standardized = standardize\_dates(dates)

print(standardized)

#Output

Error: Date 'Invalid Date' format not recognized.

['2023-01-15', '2023-12-25', '2023-03-01', '2023-04-30']

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

**dates, excluding weekends and holidays.**

from datetime import datetime, timedelta

def is\_business\_day(date):

return date.weekday() < 5 # Monday to Friday are business days (0 to 4)

def business\_days\_between(start\_date\_str, end\_date\_str, holidays=[]):

start\_date = datetime.strptime(start\_date\_str, '%Y-%m-%d')

end\_date = datetime.strptime(end\_date\_str, '%Y-%m-%d')

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

start\_date = "2023-07-01"

end\_date = "2023-07-15"

holidays = [datetime.strptime("2023-07-04", '%Y-%m-%d')] # Example of a holiday list

business\_days = business\_days\_between(start\_date, end\_date, holidays)

print(f"Business days between {start\_date} and {end\_date}: {business\_days}")

#Output  
Business days between 2023-07-01 and 2023-07-15: 9

**8. ETL Process:**

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

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

**Pandas DataFrame.**

**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.**

import pandas as pd

input\_data = [

{'id': 1, 'numeric\_field': '100', 'other\_field': 'value1'},

{'id': 2, 'numeric\_field': '200', 'other\_field': 'value2'},

{'id': 3, 'numeric\_field': 'invalid', 'other\_field': 'value3'},

{'id': 4, 'numeric\_field': 'invalid','other\_field': 'value4'}]

def extract\_transform\_data(data):

transformed\_data = []

for record in data:

try:

record['numeric\_field'] = int(record['numeric\_field'])

except ValueError:

record['numeric\_field'] = None

transformed\_data.append(record)

return transformed\_data

def validate\_data(data):

for record in data:

if 'numeric\_field' not in record or pd.isna(record['numeric\_field']):

print(f"Warning: Missing value found for record: {record}")

return True

def load\_into\_dataframe(data):

df = pd.DataFrame(data)

return df

transformed\_data = extract\_transform\_data(input\_data)

if validate\_data(transformed\_data):

df = load\_into\_dataframe(transformed\_data)

print("DataFrame successfully created:")

print(df)

else:

print("Validation failed. Data contains issues.")

**9. Data Normalization:**

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

**between 0 and 1.**

import pandas as pd

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

if column\_name not in df.columns:

raise KeyError(f"Column '{column\_name}' does not exist in the DataFrame.")

min\_val = df[column\_name].min()

max\_val = df[column\_name].max()

df\_normalized = (df[column\_name] - min\_val) / (max\_val - min\_val)

return df\_normalized

data = {

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

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

}

df = pd.DataFrame(data)

normalized\_column = normalize\_column(df, 'B')

print("Normalized column 'B':")

print(normalized\_column)

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

**standard deviation=1).**

import pandas as pd

data = {

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

}

df = pd.DataFrame(data)

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

if column\_name not in df.columns:

raise KeyError(f"Column '{column\_name}' does not exist in the DataFrame.")

mean\_val = df[column\_name].mean()

std\_val = df[column\_name].std()

df\_standardized = (df[column\_name] - mean\_val) / std\_val

return df\_standardized

standardized\_column = standardize\_column(df, 'B')

print("Standardized column 'B':")

print(standardized\_column)

**10. Advanced List Comprehensions:**

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

**numbers only, using list comprehension.**

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]

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

print("Square roots of even numbers:")

print(result)

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

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

scores = [("Alice", 85), ("Bob", 70), ("Charlie", 95), ("David", 60), ("Eve", 80)]

def names\_above\_average(scores):

total\_score = sum(score for name, 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

result = names\_above\_average(scores)

print("Students who scored above average:")

print(result)

**11.Unit Testing:**

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

**unittest framework.**

**import unittest**

def factorial(n):

if n < 0:

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

if n == 0 or n == 1:

return 1

result = 1

for i in range(2, n + 1):

result \*= i

return result

class TestFactorialFunction(unittest.TestCase):

def test\_factorial\_positive(self):

self.assertEqual(factorial(0), 1)

self.assertEqual(factorial(1), 1)

self.assertEqual(factorial(5), 120)

self.assertEqual(factorial(10), 3628800)

def test\_factorial\_negative(self):

with self.assertRaises(ValueError):

factorial(-1)

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

unittest.main()

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

def is\_palindrome(inp):

for i in range(len(inp) // 2):

if inp[i] != inp[len(inp) - i - 1]:

return False

return True

import unittest

from palindrome import is\_palindrome # Assuming is\_palindrome is defined in palindrome.py

class TestIsPalindrome(unittest.TestCase):

def test\_palindrome\_odd\_length(self):

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

def test\_palindrome\_even\_length(self):

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

def test\_not\_palindrome(self):

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

def test\_palindrome\_empty\_string(self):

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

def test\_palindrome\_single\_character(self):

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

def test\_palindrome\_case\_insensitive(self):

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

def test\_palindrome\_with\_whitespace(self):

self.assertTrue(is\_palindrome("A man a plan a canal Panama"))

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

unittest.main()

**12. Decorators:**

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

**function that sorts a large list.**

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"Function {func.\_\_name\_\_} executed in {execution\_time:.4f} seconds")

return result

return wrapper

@log\_execution\_time

def sort\_large\_list():

large\_list = list(range(1000000, 0, -1))

large\_list.sort()

sort\_large\_list()

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

**with a delay between retries.**

import time

from functools import wraps

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

def decorator(func):

@wraps(func)

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

for attempt in range(1, max\_retries + 1):

try:

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

except Exception as e:

if attempt < max\_retries:

print(f"Attempt {attempt} failed: {e}. Retrying in {delay} seconds...")

time.sleep(delay)

else:

print(f"Attempt {attempt} failed: {e}. No more retries.")

raise

return wrapper

return decorator

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

def potentially\_failing\_function():

import random

if random.random() < 0.8:

raise ValueError("Random failure")

else:

return "Success!"

result = potentially\_failing\_function()

print("Function result:", result)

**13. Concurrency with Threads:**

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

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

import threading

def sum\_part(numbers, start, end, result\_lock, result):

partial\_sum = sum(numbers[start:end])

with result\_lock:

result[0] += partial\_sum

def calculate\_sum\_with\_threads(numbers, num\_threads=4):

result = [0]

result\_lock = threading.Lock()

threads = []

chunk\_size = len(numbers) // num\_threads

for i in range(num\_threads):

start = i \* chunk\_size

end = start + chunk\_size if i < num\_threads - 1 else len(numbers)

thread = threading.Thread(target=sum\_part, args=(numbers, start, end, result\_lock, result))

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

return result[0]

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

numbers = list(range(1, 1000001)) # Example: Large list of numbers from 1 to 1,000,000

sum\_result = calculate\_sum\_with\_threads(numbers, num\_threads=4)

print(f"Sum of the numbers: {sum\_result}")

**Write a program that uses threading to fetch data from multiple URLs**

**concurrently and print the status code of each response.**

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: {e}")

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

threads = []

for url in urls:

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

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

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

urls = [

"[https://www.example.com](https://www.example.com/)",

"[https://www.google.com](https://www.google.com/)",

"[https://www.nonexistenturl.com](https://www.nonexistenturl.com/)"

]

fetch\_urls\_with\_threads(urls)

**14. Data Pipeline Simulation:**

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

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

people\_data = [

{"name": "Alice", "age": 25, "city": "New York"},

{"name": "Bob", "age": 30, "city": "San Francisco"},

{"name": "Charlie", "age": 35, "city": "Seattle"},

{"name": "David", "age": 28, "city": "Chicago"},

{"name": "Eve", "age": 22, "city": "Los Angeles"}

]

def filter\_age\_below(data, threshold):

return [person for person in data if person['age'] < threshold]

def map\_names\_to\_uppercase(data):

return [{\*\*person, "name": person["name"].upper()} for person in data]

def sort\_by\_age(data):

return sorted(data, key=lambda x: x['age'])

def data\_pipeline(data):

transformed\_data = filter\_age\_below(data, 30)

transformed\_data = map\_names\_to\_uppercase(transformed\_data)

transformed\_data = sort\_by\_age(transformed\_data)

return transformed\_data

processed\_data = data\_pipeline(people\_data)

for person in processed\_data:

print(person)

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

**encountered during processing.**

import logging

people\_data = [

{"name": "Alice", "age": 25, "city": "New York"},

{"name": "Bob", "age": 30, "city": "San Francisco"},

{"name": "Charlie", "age": 35, "city": "Seattle"},

{"name": "David", "age": 28, "city": "Chicago"},

{"name": "Eve", "city": "Los Angeles"}

]

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

def filter\_age\_below(data, threshold):

try:

filtered\_data = [person for person in data if 'age' in person and person['age'] < threshold]

return filtered\_data

except KeyError as e:

logger.error(f"KeyError in filter\_age\_below: {e}")

return []

except TypeError as e:

logger.error(f"TypeError in filter\_age\_below: {e}")

return []

except Exception as e:

logger.error(f"Unexpected error in filter\_age\_below: {e}")

return []

def map\_names\_to\_uppercase(data):

try:

mapped\_data = [{\*\*person, "name": person["name"].upper()} for person in data]

return mapped\_data

except KeyError as e:

logger.error(f"KeyError in map\_names\_to\_uppercase: {e}")

return data

except Exception as e:

logger.error(f"Unexpected error in map\_names\_to\_uppercase: {e}")

return data

def sort\_by\_age(data):

try:

sorted\_data = sorted(data, key=lambda x: x.get('age', 0))

return sorted\_data

except TypeError as e:

logger.error(f"TypeError in sort\_by\_age: {e}")

return data

except Exception as e:

logger.error(f"Unexpected error in sort\_by\_age: {e}")

return data

def handle\_errors(func):

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

try:

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

except Exception as e:

logger.error(f"Error in {func.\_\_name\_\_}: {e}")

return []

return wrapper

def data\_pipeline(data):

transformed\_data = filter\_age\_below(data, 30)

transformed\_data = map\_names\_to\_uppercase(transformed\_data)

transformed\_data = sort\_by\_age(transformed\_data)

return transformed\_data

filter\_age\_below = handle\_errors(filter\_age\_below)

map\_names\_to\_uppercase = handle\_errors(map\_names\_to\_uppercase)

sort\_by\_age = handle\_errors(sort\_by\_age)

processed\_data = data\_pipeline(people\_data)

for person in processed\_data:

print(person)

**15. Confguration Management:**

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

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

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

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

def validate\_email\_config(config):

required\_fields = ['smtp\_server', 'smtp\_port', 'sender\_email', 'receiver\_email', 'subject', 'body']

for field in required\_fields:

if field not in config:

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

value = config[field]

if not value:

return False, f"Field '{field}' is empty or has an invalid value"

if field in ['smtp\_port']:

try:

port = int(value)

if not (1 <= port <= 65535):

return False, f"Invalid value for '{field}': Port number must be between 1 and 65535"

except ValueError:

return False, f"Invalid value for '{field}': Port number must be an integer"

return True, "Configuration settings are valid"

config = {

'smtp\_server': 'smtp.example.com',

'smtp\_port': '587', # Port as a string

'sender\_email': 'lokeshramd12@gmail.com',

'receiver\_email': 'ksai11122002@gmail.com',

'subject': 'Test Email',

'body': 'This is a test email sent from Python script.'

}

valid, message = validate\_email\_config(config)

if valid:

print("Configuration settings are valid. Proceeding with task...")

# Call your task function here, e.g., send\_email(config)

else:

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

**16. Handling Large Data Sets:**

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

**the average value of the list.**

def calculate\_average\_in\_chunks(num\_list, chunk\_size):

total\_sum = 0

total\_count = 0

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

chunk = num\_list[i:i + chunk\_size]

chunk\_sum = sum(chunk)

chunk\_count = len(chunk)

total\_sum += chunk\_sum

total\_count += chunk\_count

if total\_count > 0:

average = total\_sum / total\_count

else:

average = 0

return average

numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

chunk\_size = 6

average = calculate\_average\_in\_chunks(numbers, chunk\_size)

print(f"The average of the list in chunks of size {chunk\_size} is: {average}")

**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.**

from collections import defaultdict

def count\_string\_frequency\_in\_chunks(str\_list, chunk\_size):

frequency\_dict = defaultdict(int)

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

chunk = str\_list[i:i + chunk\_size]

for string in chunk:

frequency\_dict[string] += 1

frequency\_dict = dict(frequency\_dict)

return frequency\_dict

strings = [

'apple', 'banana', 'apple', 'orange', 'apple',

'banana', 'banana', 'orange', 'apple', 'grape'

]

chunk\_size = 3

frequency\_dict = count\_string\_frequency\_in\_chunks(strings, chunk\_size)

print("Frequency dictionary:")

print(frequency\_dict)

**17. Class and Objects:**

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

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

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

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

class BankAccount:

def \_\_init\_\_(self, account\_number, balance=0.0):

self.account\_number = account\_number

self.balance = balance

def deposit(self, amount):

if amount > 0:

self.balance += amount

print(f"Deposited {amount} into account {self.account\_number}")

else:

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

def withdraw(self, amount):

if 0 < amount <= self.balance:

self.balance -= amount

print(f"Withdrew {amount} from account {self.account\_number}")

else:

print("Insufficient balance or invalid amount.")

def check\_balance(self):

print(f"Balance in account {self.account\_number}: {self.balance}")

class MultiCurrencyBankAccount(BankAccount):

def \_\_init\_\_(self, account\_number, balance=0.0, currency='USD'):

super().\_\_init\_\_(account\_number, balance)

self.currency = currency

def set\_currency(self, currency):

self.currency = currency

def convert\_to(self, currency, exchange\_rate):

if currency == self.currency:

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

return

if exchange\_rate > 0:

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

print(f"Converted {self.currency} {self.balance} to {currency} {converted\_balance}")

self.balance = converted\_balance

self.currency = currency

else:

print("Invalid exchange rate.")

def display\_balance(self):

print(f"Balance in {self.currency}: {self.balance}")

account1 = BankAccount('123456') # Account number '123456'

account1.deposit(1000)

account1.withdraw(500)

account1.check\_balance()

print("\n")

account2 = MultiCurrencyBankAccount('789012', balance=500.0, currency='USD')

account2.display\_balance()

account2.convert\_to('EUR', 0.85)

account2.display\_balance()

account2.deposit(300)

account2.display\_balance()

account2.set\_currency('GBP')

account2.convert\_to('GBP', 0.75)

account2.display\_balance()

**18. Regular Expressions:**

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

import re

def validate\_email(email):

pattern = r'^[a-zA-Z0-9\_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'

if re.match(pattern, email):

return True

else:

return False

email1 = "example.email@email.com"

email2 = "invalid\_email.com"

print(validate\_email(email1)) # True

print(validate\_email(email2)) # False

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

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

import re

def extract\_dates(text):

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

dates = re.findall(pattern, text)

return dates

text = "These are some dates: 28-02-2023, 15-10-2024, and 05-06-2025."

dates = extract\_dates(text)

print(dates)

#output

[('28', '02', '2023'), ('15', '10', '2024'), ('05', '06', '2025')]

**19. Data Encryption:**

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

**encryption from the cryptography library.**

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

**preserving the structure of the dictionary.**

from cryptography.fernet import Fernet

# Generate a key for encryption and decryption

def generate\_key():

return Fernet.generate\_key()

# Encrypt a message

def encrypt\_message(message, key):

fernet = Fernet(key)

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

return encrypted\_message

# Decrypt a message

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

fernet = Fernet(key)

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

return decrypted\_message

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

# Generate a key (this should be done once and securely stored)

key = generate\_key()

original\_message = "This is a secret message."

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

print("Encrypted:", encrypted\_message)

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

print("Decrypted:", decrypted\_message)

import json

def encrypt\_dict(dictionary, key):

json\_data = json.dumps(dictionary)

encrypted\_data = encrypt\_message(json\_data, key)

return encrypted\_data

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

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

dictionary = json.loads(decrypted\_data)

return dictionary

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

sensitive\_data = {

"username": "user123",

"password": "s3cr3tp@ssw0rd",

"credit\_card": "1234-5678-9101-1121"

}

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

print("Encrypted dictionary:", encrypted\_data)

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

print("Decrypted dictionary:", decrypted\_data)

#output

Encrypted: b'gAAAAABmoNAD2YKJR2qU7EBLPo9iHuitdpIUvM0sC0cNHJ96OtXeSajw8nryGMZDrg4bWu83-9QwLrFs4PStrwjuowgQFBEgUd-J3fVTGRARKUphVluBFsk='

Decrypted: This is a secret message.

Encrypted dictionary: b'gAAAAABmoNADSqvEoFnQxBnNamx5VVkDX39JJy96uJSfKGG0wxFiAfWx-aAZt9zuuJKp8JdklkDTbrMcIpeLqJDeGjfUkDwBvndV75y\_SM7VZEDw-b39IEwEZsa4VY1o1iwlEywj\_1Ppqe\_15xVW62zPHmb\_1SNJmoxLYT9ybXDzIHwlBGy-1YV29SnD0OZepQDsdSUahXm2'

Decrypted dictionary: {'username': 'user123', 'password': 's3cr3tp@ssw0rd', 'credit\_card': '1234-5678-9101-1121'}

**20. Memory Management:**

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

**console at regular intervals.**

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()

end\_time = start\_time + duration\_seconds

while time.time() < end\_time:

memory\_usage = psutil.virtual\_memory().used

memory\_usage\_mb = memory\_usage / (1024 \* 1024)

print(f"Memory Usage: {memory\_usage\_mb:.2f} MB")

time.sleep(interval\_seconds)

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

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

**profling to identify any memory leaks.**

import random

from memory\_profiler import profile

@profile

def generate\_large\_list(num\_elements):

large\_list = []

for i in range(num\_elements):

large\_list.append(random.randint(1, 100))

return large\_list

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

num\_elements = 10000

result = generate\_large\_list(num\_elements)

**21. Parallel Processing:**

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

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

import multiprocessing

import math

import time

def is\_prime(num):

if num <= 1:

return False

if num == 2:

return True

if num % 2 == 0:

return False

sqrt\_num = int(math.sqrt(num)) + 1

for i in range(3, sqrt\_num, 2):

if num % i == 0:

return False

return True

def find\_primes\_in\_range(start, end, result\_queue):

primes = []

for num in range(start, end):

if is\_prime(num):

primes.append(num)

result\_queue.put(primes)

def parallel\_find\_primes\_in\_range(num\_range, num\_processes):

start\_time = time.time()

chunk\_size = math.ceil((num\_range[1] - num\_range[0]) / num\_processes)

processes = []

result\_queue = multiprocessing.Queue()

for i in range(num\_processes):

start = num\_range[0] + i \* chunk\_size

end = min(num\_range[1], start + chunk\_size)

process = multiprocessing.Process(target=find\_primes\_in\_range, args=(start, end, result\_queue))

processes.append(process)

process.start()

for process in processes:

process.join()

all\_primes = []

while not result\_queue.empty():

all\_primes.extend(result\_queue.get())

end\_time = time.time()

print(f"Found {len(all\_primes)} prime numbers in range {num\_range} in {end\_time - start\_time} seconds.")

return all\_primes

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

num\_range = (1, 10000) # Define the range to find primes

num\_processes = 4 # Number of processes to use

prime\_numbers = parallel\_find\_primes\_in\_range(num\_range, num\_processes)

print("Prime numbers found:", prime\_numbers)

**Write a program that uses the multiprocessing module to perform matrix**

**multiplication in parallel.**

import multiprocessing

import numpy as np

import time

def matrix\_multiply\_chunk(A, B\_chunk, result\_queue):

C\_chunk = np.dot(A, B\_chunk)

result\_queue.put(C\_chunk)

def parallel\_matrix\_multiply(A, B, num\_processes):

start\_time = time.time()

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

B\_chunks = []

for i in range(num\_processes):

start\_col = i \* chunk\_size

end\_col = start\_col + chunk\_size if i < num\_processes - 1 else B.shape[1]

B\_chunk = B[:, start\_col:end\_col]

B\_chunks.append(B\_chunk)

result\_queue = multiprocessing.Queue()

processes = []

for B\_chunk in B\_chunks:

process = multiprocessing.Process(target=matrix\_multiply\_chunk, args=(A, B\_chunk, result\_queue))

processes.append(process)

process.start()

for process in processes:

process.join()

C\_chunks = []

while not result\_queue.empty():

C\_chunks.append(result\_queue.get())

C = np.concatenate(C\_chunks, axis=1)

end\_time = time.time()

print(f"Matrix multiplication completed in {end\_time - start\_time} seconds.")

return C

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

A = np.array([[1, 2, 3],

[4, 5, 6]])

B = np.array([[7, 8],

[9, 10],

[11, 12]])

num\_processes = 2

C = parallel\_matrix\_multiply(A, B, num\_processes)

print("Matrix A:")

print(A)

print("Matrix B:")

print(B)

print("Resultant Matrix C:")

print(C)

**22. Error Handling:**

**Write a function that raises and handles custom exceptions to manage speciϐic**

**error cases in a given task.**

def divide\_numbers(a, b):

try:

if b == 0:

raise ZeroDivisionError("Division by zero is not allowed.")

return a / b

except ZeroDivisionError as e:

raise ValueError("Cannot divide by zero.") from e

except Exception as e:

raise RuntimeError("An error occurred during division operation.") from e

try:

result = divide\_numbers(10, 0)

print("Result of division:", result)

except ValueError as ve:

print("ValueError:", ve)

except RuntimeError as re:

print("RuntimeError:", re)

#output

ERROR!

ValueError: Cannot divide by zero.

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

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

**usage.**

**23. Recursion:**

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

def fib(n):

if n <= 1:

return n

else:

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

ans=fib(3)

print(ans)

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

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

if n == 1:

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

return

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

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

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

# Example

num\_disks = 3

tower\_of\_hanoi(num\_disks, 'A', 'C', 'B')

#output

Move disk 1 from rod A to rod C

Move disk 2 from rod A to rod B

Move disk 1 from rod C to rod B

Move disk 3 from rod A to rod C

Move disk 1 from rod B to rod A

Move disk 2 from rod B to rod C

Move disk 1 from rod A to rod C

**24. Data Merging:**

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

**common key.**

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

merged\_dict = {}

merged\_list = []

for d in list1:

merged\_dict[d[key]] = d

for d in list2:

if d[key] in merged\_dict:

merged\_dict[d[key]].update(d)

else:

merged\_dict[d[key]] = d

merged\_list = list(merged\_dict.values())

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': 'San Francisco'},

{'id': 4, 'city': 'Los Angeles'},

]

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

print('Merged List of Dictionaries:')

for d in merged\_list:

print(d)

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

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

**25. Statistical Analysis:**

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

import statistics

def calculate\_statistics(numbers):

n = len(numbers)

mean = sum(numbers) / n

median = statistics.median(numbers)

mode = statistics.mode(numbers)

return mean, median, mode

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

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

print(f'List: {numbers}')

print(f'Mean: {mean}')

print(f'Median: {median}')

print(f'Mode: {mode}')

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

**numbers.**

import math

def calculate\_statistics(numbers):

n = len(numbers)

mean = sum(numbers) / n

variance = sum((x - mean) \*\* 2 for x in numbers) / n

std\_deviation = math.sqrt(variance)

return variance, std\_deviation

numbers = [30,11,14,6,28,2]

variance, std\_deviation = calculate\_statistics(numbers)

print(f'List: {numbers}')

print(f'Variance: {variance}')

print(f'Standard Deviation: {std\_deviation}')