### **Data Manipulation with Pandas:**

# Updated data with modified values

data = {

'Date': ['2024-01-01', '2024-01-02', '2024-02-01', '2024-02-02', '2024-03-01'],

'Product': ['X', 'Y', 'X', 'Y', 'X'],

'Sales': [120, 180, 240, None, 200]

}

# Convert dictionary to DataFrame

df = pd.DataFrame(data)

print("Original DataFrame:")

print(df)

# Remove rows with missing values

df.dropna(inplace=True)

print("DataFrame after removing rows with missing values:")

print(df)

# Group by 'Product' and calculate total sales for each product

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

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

print(total\_sales)

# Sort the results by total sales in descending order

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)

# Create a pivot table showing sum of sales for each product, broken down by month

df['Date'] = pd.to\_datetime(df['Date'])

df['Month'] = df['Date'].dt.month\_name()

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

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

print(pivot\_table)

2. Data Cleaning:

# Updated data with modified values

data = {

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

'Name': ['Mike', 'Sara', 'Oliver', 'Emma', 'Liam'],

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

'Score': [85, 92, np.nan, 78, 88],

'Join\_Date': ['2023-01-01', '2022-03-15', '', '2023-05-20', '2022-12-10']

}

df = pd.DataFrame(data)

print(df)

# Function to clean the DataFrame

def clean\_dataframe(df):

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

numeric\_columns = df.select\_dtypes(include=[np.number]).columns

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

text\_columns = df.select\_dtypes(include=[object]).columns

df[text\_columns] = df[text\_columns].apply(lambda x: x.str.lower() if x.dtype == "object" else x)

return df

# Function to remove outliers from a numeric column using IQR method

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

q1 = df[column\_name].quantile(0.25)

q3 = df[column\_name].quantile(0.75)

iqr = q3 - q1

lower\_bound = q1 - 1.5 \* iqr

upper\_bound = q3 + 1.5 \* iqr

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

return df\_filtered

df['Join\_Date'] = pd.to\_datetime(df['Join\_Date'], errors='coerce')

df['Month'] = df['Join\_Date'].dt.month\_name()

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

print("Pivot table showing sum of 'Score' for each 'Name', broken down by 'Month':")

print(pivot\_table)

cleaned\_df = clean\_dataframe(df)

print("DataFrame after cleaning:")

print(cleaned\_df)

score\_outliers\_removed = remove\_outliers(cleaned\_df, 'Score')

print("DataFrame after removing outliers from 'Score' column:")

print(score\_outliers\_removed)

3. Lambda Functions and Map-Reduce:

# Filter out even numbers using lambda function

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

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

print(filtered\_numbers)

# Calculate the product of remaining numbers using reduce function

from functools import reduce

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

print("Product of remaining numbers:", product)

# Remove words shorter than 4 characters using filter and concatenate remaining words using reduce

words = ['apple', 'banana', 'grape', 'kiwi', 'melon']

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

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

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

4. Data Visualization:

# Line chart showing sales trend over time

import pandas as pd

import matplotlib.pyplot as plt

data = {'Date': pd.date\_range(start='2023-01-01', periods=10, freq='M'),

'Sales': [100, 120, 130, 110, 150, 140, 160, 180, 170, 190]}

df = pd.DataFrame(data)

plt.figure(figsize=(10, 6))

plt.plot(df['Date'], df['Sales'], marker='o', linestyle='-', color='b')

plt.title('Sales Trend Over Time')

plt.xlabel('Date')

plt.ylabel('Sales')

plt.show()

# Scatter plot with trend line

import numpy as np

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')

# Adding a trend line (best-fit line)

m, b = np.polyfit(x, y, 1)

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

plt.legend()

plt.show()

5. Data Aggregation:

# Aggregate total amount spent by each user

def aggregate\_total\_amount(transactions):

total\_amounts = {}

for transaction in transactions:

user = transaction['user']

amount = transaction['amount']

if user in total\_amounts:

total\_amounts[user] += amount

else:

total\_amounts[user] = amount

return total\_amounts

# Calculate moving average of total amount spent by each user over a specified window size

def moving\_average\_total\_amount(total\_amounts, window\_size):

moving\_averages = {}

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

moving\_avg = []

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

moving\_averages[user] = None

else:

for i in range(len(total\_amounts[user]) - window\_size + 1):

window\_sum = sum(total\_amounts[user][i:i+window\_size])

window\_avg = window\_sum / window\_size

moving\_avg.append(window\_avg)

moving\_averages[user] = moving\_avg

return moving\_averages

transactions = [

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

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

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

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

{'user': 'Jane', 'amount': 120},

{'user': 'John', 'amount': 80}

]

total\_amounts = aggregate\_total\_amount(transactions)

print("Total amounts spent by each user:")

print(total\_amounts)

window\_size = 3

moving\_averages = moving\_average\_total\_amount(total\_amounts, window\_size)

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

print(moving\_averages)

6. Exception Handling:

# Function to safely divide two numbers

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)) # Output: 5.0

print(safe\_divide(5, 0))

# Function to open multiple files and handle exceptions

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:

# Function to standardize date format in a list of dates

from datetime import datetime

def standardize\_dates(dates):

standardized\_dates = []

for date\_str in dates:

try:

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

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

except ValueError:

standardized\_dates.append('Invalid Date Format')

return standardized\_dates

dates = ['2023-01-15', '2023-05-30', '2023/03/25', '2022-12-10']

standardized\_dates = standardize\_dates(dates)

print("Standardized dates:")

print(standardized\_dates)

# Function to find the most recent date in a list of dates

def find\_recent\_date(dates):

recent\_date = None

for date\_str in dates:

try:

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

if recent\_date is None or date > recent\_date:

recent\_date = date

except ValueError:

pass

return recent\_date.strftime('%d-%m-%Y') if recent\_date is not None else 'No valid dates'

dates = ['2023-01-15', '2023-05-30', '2023/03/25', '2022-12-10']

recent\_date = find\_recent\_date(dates)

print("Most recent date:", recent\_date)

'''8. ETL Process:

· 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.

· 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

from sklearn.preprocessing import MinMaxScaler

import numpy as np

data = [

{"id": 1, "name": "Alice", "age": 34},

{"id": 2, "name": "Bob", "age": 45},

{"id": 3, "name": "Charlie", "age": 23},

{"id": 4, "name": "David", "age": 29},

{"id": 5, "name": None, "age": 50},

{"id": 6, "name": "Eve", "age": None},

{"id": 7, "name": "Frank", "age": 30}

]

df=pd.DataFrame(data)

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

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

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

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

return df

def validate\_data(df):

issues = []

if df.isnull().values.any():

issues.append("Data contains missing values.")

numeric\_df = df.select\_dtypes(include=['int64', 'float64'])

z\_scores = (numeric\_df - numeric\_df.mean()) / numeric\_df.std()

if ((z\_scores.abs() > 3).any().any()):

issues.append("Data contains outliers.")

return issues

def etl\_process(data):

extracted\_data = pd.DataFrame(data)

validation\_issues = validate\_data(extracted\_data)

if validation\_issues:

print("Validation issues found:")

for issue in validation\_issues:

print(f" - {issue}")

return None

transformed\_data = normalize\_data(extracted\_data,'age')

df = pd.DataFrame(transformed\_data)

return df

df = etl\_process(data)

if df is not None:

print(df)

'''9. Data Normalization:

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

between 0 and 1.

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

standard deviation=1).'''

import pandas as pd

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

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

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

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

return df

data = {

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

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

}

df = pd.DataFrame(data)

print("Original DataFrame:")

print(df)

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

print("\nNormalized DataFrame:")

print(df\_normalized)

import pandas as pd

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

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

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

df[column\_name] = (df[column\_name] - mean\_value) / std\_dev

return df

data = {

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

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

}

df = pd.DataFrame(data)

print("Original DataFrame:")

print(df)

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

print("\nStandardized DataFrame:")

print(df\_standardized)

'''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

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

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

print( even\_roots)

'''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", 75), ("Charlie", 95), ("David", 65)]

average\_score = sum(score for name, score in scores) / len(scores)

names\_above\_average = [name for name, score in scores if score > average\_score]

print("Names of students who scored above average:", names\_above\_average)

'''11. Unit Testing:'''

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

unittest framework.'''

def factorial(n):

if n < 0:

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

elif n == 0 or n == 1:

return 1

else:

result = 1

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

result \*= i

return result

import unittest

class TestFactorial(unittest.TestCase):

def test\_factorial\_zero(self):

self.assertEqual(factorial(0), 1)

def test\_factorial\_one(self):

self.assertEqual(factorial(1), 1)

def test\_factorial\_positive(self):

self.assertEqual(factorial(5), 120)

self.assertEqual(factorial(3), 6)

self.assertEqual(factorial(10), 3628800)

def test\_factorial\_negative(self):

with self.assertRaises(ValueError):

factorial(-1)

def test\_factorial\_large(self):

self.assertEqual(factorial(20), 2432902008176640000)

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

unittest.main()

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

def is\_palindrome(s):

s = ''.join(c.lower() for c in s if c.isalnum()) # Normalize the string

return s == s[::-1]

class TestIsPalindrome(unittest.TestCase):

def test\_empty\_string(self):

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

def test\_single\_character(self):

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

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

def test\_simple\_palindrome(self):

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

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

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

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

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

def test\_palindrome\_with\_non\_alphanumeric(self):

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

self.assertTrue(is\_palindrome("No 'x' in Nixon"))

def test\_non\_palindrome(self):

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

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

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

unittest.main()

'''12. Decorators:'''

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

function that sorts a large list.'''

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

with a delay between retries.'''

import time

import functools

import random

# Decorator for logging execution time

def log\_execution\_time(func):

@functools.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"Executed {func.\_\_name\_\_} in {execution\_time:.4f} seconds")

return result

return wrapper

# Function to sort a large list

@log\_execution\_time

def sort\_large\_list(lst):

return sorted(lst)

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

def decorator(func):

@functools.wraps(func)

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

retries = 0

while retries < max\_retries:

try:

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

except Exception as e:

retries += 1

print(f"Exception: {e}. Retrying {retries}/{max\_retries}...")

time.sleep(delay)

raise Exception(f"Failed after {max\_retries} retries")

return wrapper

return decorator

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

def unreliable\_function():

if random.random() < 0.5: # 50% chance to raise an exception

raise ValueError("Random failure occurred")

return "Success"

# Example usage

large\_list = list(range(100, 0, -1))

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

print(sorted\_list)

try:

result = unreliable\_function()

print(result)

except Exception as e:

print(e)

'''13.. Concurrency with Threads:'''

'''a.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 partial\_sum(numbers, start, end, result, index):

result[index] = sum(numbers[start:end])

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

length = len(numbers)

segment\_size = length // num\_threads

threads = []

result = [0] \* num\_threads

for i in range(num\_threads):

start = i \* segment\_size

end = (i + 1) \* segment\_size if i != num\_threads - 1 else length

thread = threading.Thread(target=partial\_sum, args=(numbers, start, end, result, i))

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

return sum(result)

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

large\_list = list(range(1, 10000001))

num\_threads = 4

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

print("Total sum:", total\_sum)

'''b.Write a program that uses threading to fetch data from multiple URLs

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

pip install requests

import threading

import requests

def fetch\_url(url, results, index):

try:

response = requests.get(url)

results[index] = (url, response.status\_code)

except requests.RequestException as e:

results[index] = (url, None, str(e))

def fetch\_data\_from\_urls(urls):

num\_urls = len(urls)

threads = []

results = [None] \* num\_urls

for i in range(num\_urls):

thread = threading.Thread(target=fetch\_url, args=(urls[i], results, i))

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

for result in results:

if result[1] is not None:

print(f"URL: {result[0]}, Status Code: {result[1]}")

else:

print(f"URL: {result[0]}, Error: {result[2]}")

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

urls = [

"https://www.example.com",

"https://www.google.com",

"https://www.nonexistentwebsite123456.com",

"https://www.github.com"

]

fetch\_data\_from\_urls(urls)

'''14. Data Pipeline Simulation:'''

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

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

def add\_new\_field(data, field\_name, default\_value):

for record in data:

record[field\_name] = default\_value

return data

def normalize\_field(data, field\_name):

values = [record[field\_name] for record in data]

min\_value = min(values)

max\_value = max(values)

range\_value = max\_value - min\_value

for record in data:

record[field\_name] = (record[field\_name] - min\_value) / range\_value

return data

def convert\_field\_to\_uppercase(data, field\_name):

for record in data:

if field\_name in record:

record[field\_name] = record[field\_name].upper()

return data

def filter\_records(data, field\_name, threshold):

return [record for record in data if record.get(field\_name, 0) > threshold]

def process\_data\_pipeline(data, transformations):

for transformation in transformations:

data = transformation(data)

return data

# Example usage

data = [

{'name': 'Alice', 'age': 25, 'score': 85},

{'name': 'Bob', 'age': 30, 'score': 90},

{'name': 'Charlie', 'age': 35, 'score': 75},

{'name': 'David', 'age': 40, 'score': 60}

]

transformations = [

lambda data: add\_new\_field(data, 'status', 'active'),

lambda data: normalize\_field(data, 'score'),

lambda data: convert\_field\_to\_uppercase(data, 'name'),

lambda data: filter\_records(data, 'age', 30)

]

processed\_data = process\_data\_pipeline(data, transformations)

print("Processed Data:")

for record in processed\_data:

print(record)

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

encountered during processing'''

import logging

logging.basicConfig(filename='pipeline\_errors.log', level=logging.ERROR)

def log\_error(function\_name, error):

logging.error(f"Error in function {function\_name}: {error}")

def add\_new\_field(data, field\_name, default\_value):

try:

for record in data:

record[field\_name] = default\_value

except Exception as e:

log\_error('add\_new\_field', e)

return data

def normalize\_field(data, field\_name):

try:

values = [record[field\_name] for record in data]

min\_value = min(values)

max\_value = max(values)

range\_value = max\_value - min\_value

for record in data:

record[field\_name] = (record[field\_name] - min\_value) / range\_value

except Exception as e:

log\_error('normalize\_field', e)

return data

def convert\_field\_to\_uppercase(data, field\_name):

try:

for record in data:

if field\_name in record:

record[field\_name] = record[field\_name].upper()

except Exception as e:

log\_error('convert\_field\_to\_uppercase', e)

return data

def filter\_records(data, field\_name, threshold):

try:

return [record for record in data if record.get(field\_name, 0) > threshold]

except Exception as e:

log\_error('filter\_records', e)

return data # Return the original data if an error occurs

def process\_data\_pipeline(data, transformations):

for transformation in transformations:

try:

data = transformation(data)

except Exception as e:

log\_error('process\_data\_pipeline', e)

return data

# Example usage

data = [

{'name': 'Alice', 'age': 25, 'score': 85},

{'name': 'Bob', 'age': 30, 'score': 90},

{'name': 'Charlie', 'age': 35, 'score': 75},

{'name': 'David', 'age': 40, 'score': 60}

]

transformations = [

lambda data: add\_new\_field(data, 'status', 'active'),

lambda data: normalize\_field(data, 'score'),

lambda data: convert\_field\_to\_uppercase(data, 'name'),

lambda data: filter\_records(data, 'age', 30)

]

processed\_data = process\_data\_pipeline(data, transformations)

print("Processed Data:")

for record in processed\_data:

print(record)

'''15. Conϐiguration Management:'''

'''a.Write a Python script that reads conϐiguration settings from a dictionary and

uses them to perform a speciϐic task.'''

config = {

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

"smtp\_port": 587,

"username": "user@example.com",

"password": "password",

"from\_email": "user@example.com",

"to\_email": "recipient@example.com",

"subject": "Test Email",

"body": "This is a test email sent from Python."

}

import smtplib

from email.mime.text import MIMEText

from email.mime.multipart import MIMEMultipart

def send\_email(config):

try:

# Extract settings from the configuration dictionary

smtp\_server = config["smtp\_server"]

smtp\_port = config["smtp\_port"]

username = config["username"]

password = config["password"]

from\_email = config["from\_email"]

to\_email = config["to\_email"]

subject = config["subject"]

body = config["body"]

# Create the email message

msg = MIMEMultipart()

msg['From'] = from\_email

msg['To'] = to\_email

msg['Subject'] = subject

msg.attach(MIMEText(body, 'plain'))

# Connect to the SMTP server and send the email

with smtplib.SMTP(smtp\_server, smtp\_port) as server:

server.starttls() # Upgrade the connection to a secure encrypted SSL/TLS connection

server.login(username, password)

server.send\_message(msg)

print("Email sent successfully!")

except Exception as e:

print(f"An error occurred: {e}")

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

# Configuration dictionary

config = {

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

"smtp\_port": 587,

"username": "user@example.com",

"password": "password",

"from\_email": "user@example.com",

"to\_email": "recipient@example.com",

"subject": "Test Email",

"body": "This is a test email sent from Python."

}

send\_email(config)

'''b.Write a function that validates the conϐiguration settings, ensuring that all

required ϐields are present and have valid values.'''

import re

def validate\_config(config):

required\_fields = {

"smtp\_server": str,

"smtp\_port": int,

"username": str,

"password": str,

"from\_email": str,

"to\_email": str,

"subject": str,

"body": str

}

for field, expected\_type in required\_fields.items():

if field not in config:

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

if not isinstance(config[field], expected\_type):

return False, f"Field '{field}' must be of type {expected\_type.\_\_name\_\_}"

if not (1 <= config["smtp\_port"] <= 65535):

return False, "SMTP port must be between 1 and 65535"

email\_pattern = re.compile(r"^[\w\.-]+@[\w\.-]+\.\w+$")

if not email\_pattern.match(config["from\_email"]):

return False, "Invalid 'from\_email' address format"

if not email\_pattern.match(config["to\_email"]):

return False, "Invalid 'to\_email' address format"

return True, "Configuration is valid"

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

config = {

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

"smtp\_port": 587,

"username": "user@example.com",

"password": "password",

"from\_email": "user@example.com",

"to\_email": "recipient@example.com",

"subject": "Test Email",

"body": "This is a test email sent from Python."

}

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

if is\_valid:

print("Configuration is valid.")

else:

print(f"Configuration is invalid: {message}")

'''16. Handling Large Data Sets:'''

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

the average value of the list.'''

def calculate\_average\_large\_list(numbers, chunk\_size=10000):

total\_sum = 0

total\_count = 0

iterator = iter(numbers)

while True:

# Get the next chunk

chunk = list(itertools.islice(iterator, chunk\_size))

if not chunk:

break

chunk\_sum = sum(chunk)

chunk\_count = len(chunk)

total\_sum += chunk\_sum

total\_count += chunk\_count

if total\_count == 0:

return 0

return total\_sum / total\_count

numbers = range(1, 1000000)

average = calculate\_average\_large\_list(numbers)

print(f"The average is {average}")

'''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.'''

from collections import defaultdict

import itertools

def count\_frequencies\_large\_list(strings, chunk\_size=10000):

frequency\_dict = defaultdict(int)

iterator = iter(strings)

while True:

chunk = list(itertools.islice(iterator, chunk\_size))

if not chunk:

break

for string in chunk:

frequency\_dict[string] += 1

return dict(frequency\_dict)

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

frequency = count\_frequencies\_large\_list(strings)

print(frequency)

'''17.Class and Objects:'''

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

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

b.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, initial\_balance=0):

if initial\_balance < 0:

raise ValueError("Initial balance cannot be negative.")

self.balance = initial\_balance

def deposit(self, amount):

if amount <= 0:

raise ValueError("Deposit amount must be positive.")

self.balance += amount

def withdraw(self, amount):

if amount <= 0:

raise ValueError("Withdrawal amount must be positive.")

if amount > self.balance:

raise ValueError("Insufficient funds.")

self.balance -= amount

def check\_balance(self):

return self.balance

# Example usage:

account = BankAccount(100)

account.deposit(50)

print(account.check\_balance()) # Output: 150

account.withdraw(30)

print(account.check\_balance()) # Output: 120

class CurrencyBankAccount:

def \_\_init\_\_(self, initial\_balance=0, currency='USD'):

if initial\_balance < 0:

raise ValueError("Initial balance cannot be negative.")

self.balance = initial\_balance

self.currency = currency

self.exchange\_rates = {} # Dictionary to hold exchange rates

def set\_exchange\_rate(self, from\_currency, to\_currency, rate):

if rate <= 0:

raise ValueError("Exchange rate must be positive.")

self.exchange\_rates[(from\_currency, to\_currency)] = rate

def convert\_currency(self, amount, from\_currency, to\_currency):

if from\_currency == to\_currency:

return amount

if (from\_currency, to\_currency) not in self.exchange\_rates:

raise ValueError("Exchange rate not set.")

return amount \* self.exchange\_rates[(from\_currency, to\_currency)]

def deposit(self, amount, currency):

if amount <= 0:

raise ValueError("Deposit amount must be positive.")

if currency != self.currency:

amount = self.convert\_currency(amount, currency, self.currency)

self.balance += amount

def withdraw(self, amount, currency):

if amount <= 0:

raise ValueError("Withdrawal amount must be positive.")

if currency != self.currency:

amount = self.convert\_currency(amount, currency, self.currency)

if amount > self.balance:

raise ValueError("Insufficient funds.")

self.balance -= amount

def check\_balance(self):

return self.balance

# Example usage:

account = CurrencyBankAccount(100, 'USD')

account.set\_exchange\_rate('USD', 'EUR', 0.85)

account.set\_exchange\_rate('EUR', 'USD', 1.18)

account.deposit(50, 'EUR')

print(account.check\_balance()) # Output: 100 + (50 \* 1.18) = 159.0

account.withdraw(30, 'USD')

print(account.check\_balance()) # Output: 159.0 - 30 = 129.0

'''18.Regular Expressions:'''

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

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

(DD-MM-YYYY).'''

import re

def validate\_email(email):

# Regular expression for validating an Email

pattern = r'^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

if re.match(pattern, email):

return True

return False

# Example usage:

print(validate\_email('example@example.com')) # Output: True

print(validate\_email('invalid-email@com')) # Output: False

import re

def extract\_dates(text):

# Regular expression for matching dates in DD-MM-YYYY format

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

return re.findall(pattern, text)

# Example usage:

text = "The event is on 15-08-2024 and the deadline is 31-12-2024."

dates = extract\_dates(text)

print(dates) # Output: ['15-08-2024', '31-12-2024']

'''19'''

from cryptography.fernet import Fernet

def generate\_key():

"""Generate a new Fernet key."""

return Fernet.generate\_key()

def encrypt\_text(key, text):

"""Encrypt the given text using the provided key."""

fernet = Fernet(key)

encrypted\_text = fernet.encrypt(text.encode())

return encrypted\_text

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

"""Decrypt the given encrypted text using the provided key."""

fernet = Fernet(key)

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

return decrypted\_text

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

key = generate\_key()

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

text = "This is a secret message."

print(f"Original Text: {text}")

encrypted = encrypt\_text(key, text)

print(f"Encrypted Text: {encrypted.decode()}")

decrypted = decrypt\_text(key, encrypted)

print(f"Decrypted Text: {decrypted}")

from cryptography.fernet import Fernet

import json

def generate\_key():

"""Generate a new Fernet key."""

return Fernet.generate\_key()

def encrypt\_text(key, text):

"""Encrypt the given text using the provided key."""

fernet = Fernet(key)

encrypted\_text = fernet.encrypt(text.encode())

return encrypted\_text

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

"""Decrypt the given encrypted text using the provided key."""

fernet = Fernet(key)

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

return decrypted\_text

def encrypt\_dict(key, data):

"""Encrypt all string values in a dictionary, preserving its structure."""

encrypted\_data = {}

for k, v in data.items():

if isinstance(v, dict):

encrypted\_data[k] = encrypt\_dict(key, v)

elif isinstance(v, str):

encrypted\_data[k] = encrypt\_text(key, v)

else:

encrypted\_data[k] = v

return encrypted\_data

def decrypt\_dict(key, data):

"""Decrypt all string values in a dictionary, preserving its structure."""

decrypted\_data = {}

for k, v in data.items():

if isinstance(v, dict):

decrypted\_data[k] = decrypt\_dict(key, v)

elif isinstance(v, bytes):

decrypted\_data[k] = decrypt\_text(key, v)

else:

decrypted\_data[k] = v

return decrypted\_data

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

key = generate\_key()

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

sensitive\_data = {

"username": "user123",

"password": "securepassword",

"details": {

"email": "user123@example.com",

"phone": "123-456-7890"

}

}

print(f"Original Data: {json.dumps(sensitive\_data, indent=4)}")

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

print(f"Encrypted Data: {json.dumps(encrypted\_data, indent=4)}")

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

print(f"Decrypted Data: {json.dumps(decrypted\_data, indent=4)}")

'''20'''

import psutil

import time

import os

def monitor\_memory(pid, interval=1):

"""Monitor and log memory usage of the process with the given PID."""

try:

process = psutil.Process(pid)

while True:

mem\_info = process.memory\_info()

print(f"Memory Usage: {mem\_info.rss / 1024 / 1024:.2f} MB")

time.sleep(interval)

except psutil.NoSuchProcess:

print(f"No process found with PID: {pid}")

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

pid = os.getpid() # Get current process ID

monitor\_memory(pid, interval=2)

import random

from memory\_profiler import profile

@profile

def generate\_large\_list(size):

"""Generate a large list of random numbers."""

large\_list = [random.randint(1, 100) for \_ in range(size)]

return large\_list

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

generate\_large\_list(10\*\*6) # Adjust the size as needed

'''21'''

import multiprocessing

def is\_prime(num):

"""Check if a number is prime."""

if num <= 1:

return False

if num <= 3:

return True

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

return False

i = 5

while i \* i <= num:

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

return False

i += 6

return True

def calculate\_primes(start, end, result\_queue):

"""Calculate prime numbers in a range and put them in a result queue."""

primes = [num for num in range(start, end) if is\_prime(num)]

result\_queue.put(primes)

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

"""Parallelize the prime number calculation across multiple processes."""

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

processes = []

result\_queue = multiprocessing.Queue()

for i in range(num\_processes):

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

chunk\_end = start + (i + 1) \* chunk\_size if i < num\_processes - 1 else end

process = multiprocessing.Process(target=calculate\_primes, args=(chunk\_start, chunk\_end, result\_queue))

processes.append(process)

process.start()

for process in processes:

process.join()

primes = []

while not result\_queue.empty():

primes.extend(result\_queue.get())

return sorted(primes)

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

start = 1

end = 100000

num\_processes = multiprocessing.cpu\_count()

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

print(f"Number of primes found: {len(primes)}")

import multiprocessing

import numpy as np

def matrix\_multiply\_chunk(A, B, result, row\_start, row\_end):

"""Multiply a chunk of rows from matrix A with matrix B and store in the result matrix."""

for i in range(row\_start, row\_end):

for j in range(B.shape[1]):

result[i, j] = sum(A[i, k] \* B[k, j] for k in range(A.shape[1]))

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

"""Perform matrix multiplication in parallel using multiple processes."""

result = np.zeros((A.shape[0], B.shape[1]), dtype=int)

chunk\_size = A.shape[0] // num\_processes

processes = []

for i in range(num\_processes):

row\_start = i \* chunk\_size

row\_end = (i + 1) \* chunk\_size if i < num\_processes - 1 else A.shape[0]

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

processes.append(process)

process.start()

for process in processes:

process.join()

return result

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

# Example matrices

A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

B = np.array([[9, 8, 7], [6, 5, 4], [3, 2, 1]])

num\_processes = multiprocessing.cpu\_count()

result = parallel\_matrix\_multiplication(A, B, num\_processes)

print("Matrix A:")

print(A)

print("Matrix B:")

print(B)

print("Result of A x B:")

print(result)

'''22'''

class DivisionByZeroError(Exception):

"""Exception raised for errors in the division by zero."""

def \_\_init\_\_(self, message="Cannot divide by zero"):

self.message = message

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

class NegativeNumberError(Exception):

"""Exception raised for errors in operations with negative numbers."""

def \_\_init\_\_(self, message="Negative number encountered"):

self.message = message

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

def divide\_numbers(numerator, denominator):

"""Divide two numbers and handle custom exceptions."""

try:

if denominator == 0:

raise DivisionByZeroError()

if numerator < 0 or denominator < 0:

raise NegativeNumberError()

result = numerator / denominator

return result

except DivisionByZeroError as e:

print(f"Error: {e}")

except NegativeNumberError as e:

print(f"Error: {e}")

except Exception as e:

print(f"An unexpected error occurred: {e}")

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

# Test cases

print(divide\_numbers(10, 2)) # Should work fine

print(divide\_numbers(10, 0)) # Should raise DivisionByZeroError

print(divide\_numbers(-10, 2)) # Should raise NegativeNumberError

from contextlib import contextmanager

@contextmanager

def database\_connection():

"""Context manager for handling a mock database connection."""

connection = None

try:

# Simulate opening a database connection

connection = "Database Connection Established"

print(connection)

yield connection # Provide the resource to the caller

except Exception as e:

print(f"Error during database operation: {e}")

raise # Re-raise the exception to be handled outside if needed

finally:

if connection:

# Simulate closing the database connection

print("Database Connection Closed")

def perform\_database\_operation():

"""Function to perform a database operation using the context manager."""

try:

with database\_connection() as conn:

# Simulate a database operation

print("Performing database operation...")

# Simulate an error during operation

raise ValueError("Simulated database error")

except ValueError as e:

print(f"Handled ValueError: {e}")

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

perform\_database\_operation()

'''23'''

def fibonacci(n):

if n <= 0:

raise ValueError("n must be a positive integer")

elif n == 1:

return 0

elif n == 2:

return 1

else:

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

print(fibonacci(10))

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

if n <= 0:

raise ValueError("n must be a positive integer")

if n == 1:

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

else:

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

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

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

tower\_of\_hanoi(3, 'A', 'B', 'C')

'''24'''

def merge\_dict\_lists(list1, list2, key):

merged\_dict = {}

# Add dictionaries from the first list

for d in list1:

merged\_dict[d[key]] = d

# Update with dictionaries from the second list

for d in list2:

if d[key] in merged\_dict:

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

else:

merged\_dict[d[key]] = d

return list(merged\_dict.values())

# Example usage

list1 = [

{'id': 1, 'name': 'Alice', 'age': 30},

{'id': 2, 'name': 'Bob', 'age': 25}

]

list2 = [

{'id': 1, 'city': 'New York'},

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

]

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

print(merged\_list)

import pandas as pd

def merge\_dataframes(dfs, key):

# Start with the first DataFrame

merged\_df = dfs[0]

# Merge each DataFrame in the list

for df in dfs[1:]:

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

# Resolve conflicts by keeping the most recent data

for column in df.columns:

if column != key:

if column + '\_new' in merged\_df.columns:

merged\_df[column] = merged\_df.apply(lambda row: row[column + '\_new'] if pd.notnull(row[column + '\_new']) else row[column], axis=1)

merged\_df.drop(column + '\_new', axis=1, inplace=True)

return merged\_df

# Example usage

df1 = pd.DataFrame({

'id': [1, 2],

'name': ['Alice', 'Bob'],

'age': [30, 25]

})

df2 = pd.DataFrame({

'id': [1, 3],

'city': ['New York', 'Los Angeles'],

'age': [35, 40] # New data for id 1, new entry for id 3

})

df3 = pd.DataFrame({

'id': [2, 3],

'age': [26, 41], # New data for id 2 and id 3

'country': ['USA', 'USA']

})

dataframes = [df1, df2, df3]

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

print(merged\_df)

'''25'''

from statistics import mean, median, mode, StatisticsError

def calculate\_mean\_median\_mode(numbers):

if not numbers:

raise ValueError("The list is empty")

mean\_value = mean(numbers)

median\_value = median(numbers)

try:

mode\_value = mode(numbers)

except StatisticsError:

mode\_value = None # No unique mode in the list

return mean\_value, median\_value, mode\_value

from statistics import stdev, variance

def calculate\_standard\_deviation\_variance(numbers):

if len(numbers) < 2:

raise ValueError("At least two numbers are required")

stdev\_value = stdev(numbers)

variance\_value = variance(numbers)

return stdev\_value, variance\_value

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

mean\_value, median\_value, mode\_value = calculate\_mean\_median\_mode(numbers)

print(f"Mean: {mean\_value}, Median: {median\_value}, Mode: {mode\_value}")

stdev\_value, variance\_value = calculate\_standard\_deviation\_variance(numbers)

print(f"Standard Deviation: {stdev\_value}, Variance: {variance\_value}")