1. **Implement a Python class MaxHeap that supports the following operations: insert, delete, and get\_max. Ensure the operations maintain the properties of a max-heap.**

class MaxHeap:

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

self.heap = []

def insert(self, value):

self.heap.append(value)

self.\_heapify\_up(len(self.heap) - 1)

def delete(self):

if len(self.heap) == 0:

raise IndexError("delete from an empty heap")

if len(self.heap) == 1:

return self.heap.pop()

max\_value = self.heap[0]

self.heap[0] = self.heap.pop()

self.\_heapify\_down(0)

return max\_value

def get\_max(self):

if len(self.heap) == 0:

raise IndexError("get from an empty heap")

return self.heap[0]

def \_heapify\_up(self, index):

parent\_index = (index - 1) // 2

if index > 0 and self.heap[index] > self.heap[parent\_index]:

self.heap[index], self.heap[parent\_index] = self.heap[parent\_index], self.heap[index]

self.\_heapify\_up(parent\_index)

def \_heapify\_down(self, index):

largest = index

left\_child\_index = 2 \* index + 1

right\_child\_index = 2 \* index + 2

if left\_child\_index < len(self.heap) and self.heap[left\_child\_index] > self.heap[largest]:

largest = left\_child\_index

if right\_child\_index < len(self.heap) and self.heap[right\_child\_index] > self.heap[largest]:

largest = right\_child\_index

if largest != index:

self.heap[index], self.heap[largest] = self.heap[largest], self.heap[index]

self.\_heapify\_down(largest)

def \_\_str\_\_(self):

return str(self.heap)

# Example usage:

heap = MaxHeap()

heap.insert(10)

heap.insert(20)

heap.insert(5)

heap.insert(7)

heap.insert(30)

print(heap) # Output: [30, 10, 20, 5, 7]

print(heap.get\_max()) # Output: 30

heap.delete() # Removes the maximum element (30)

print(heap) # Output: [20, 10, 7, 5]

1. **Write a Python function that takes a list of URLs, attempts to download their content, and retries up to 3 times if an error occurs. Use appropriate error handling to manage different types of exceptions.**

import requests

from requests.exceptions import HTTPError, ConnectionError, Timeout, RequestException

def download\_content(urls):

contents = {}

for url in urls:

attempts = 0

success = False

while attempts < 3 and not success:

try:

response = requests.get(url, timeout=10)

response.raise\_for\_status()

contents[url] = response.text

success = True

except HTTPError as http\_err:

print(f"HTTP error occurred: {http\_err} - URL: {url}")

except ConnectionError as conn\_err:

print(f"Connection error occurred: {conn\_err} - URL: {url}")

except Timeout as timeout\_err:

print(f"Timeout error occurred: {timeout\_err} - URL: {url}")

except RequestException as req\_err:

print(f"Error occurred: {req\_err} - URL: {url}")

finally:

attempts += 1

if not success and attempts < 3:

print(f"Retrying {url} (attempt {attempts + 1})...")

if not success:

contents[url] = None

print(f"Failed to retrieve {url} after 3 attempts")

return contents

# Example usage:

urls = [

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

"https://www.nonexistentwebsite.com", # This will fail

"https://www.google.com"

]

result = download\_content(urls)

for url, content in result.items():

if content:

print(f"Content from {url[:30]}...: {content[:100]}") # Print first 100 characters of the content

else:

print(f"No content retrieved from {url}")

1. **Write a Python script that trains a simple linear regression model using scikit-learn. Use a dataset of your choice, split it into training and testing sets, and evaluate the model's performance.**

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.datasets import load\_diabetes

from sklearn.preprocessing import StandardScaler

# Load the diabetes dataset

diabetes = load\_diabetes()

X = pd.DataFrame(diabetes.data, columns=diabetes.feature\_names)

y = pd.Series(diabetes.target)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Standardize the numerical columns

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# Create the model

model = LinearRegression()

# Train the model

model.fit(X\_train\_scaled, y\_train)

# Make predictions

y\_train\_pred = model.predict(X\_train\_scaled)

y\_test\_pred = model.predict(X\_test\_scaled)

# Evaluate the model

train\_mse = mean\_squared\_error(y\_train, y\_train\_pred)

test\_mse = mean\_squared\_error(y\_test, y\_test\_pred)

train\_r2 = r2\_score(y\_train, y\_train\_pred)

test\_r2 = r2\_score(y\_test, y\_test\_pred)

print("Training set evaluation:")

print(f"Mean Squared Error: {train\_mse:.2f}")

print(f"R^2 Score: {train\_r2:.2f}")

print("\nTesting set evaluation:")

print(f"Mean Squared Error: {test\_mse:.2f}")

print(f"R^2 Score: {test\_r2:.2f}")

# Display model coefficients

print("\nModel coefficients:")

coefficients = pd.DataFrame(model.coef\_, X.columns, columns=['Coefficient'])

print(coefficients)

1. **Using pandas, write a Python function to clean and preprocess a given DataFrame, which involves handling missing values, normalizing numerical columns, and encoding categorical columns.**

import pandas as pd

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

from sklearn.impute import SimpleImputer

from sklearn.pipeline import Pipeline

def preprocess\_data(df):

# Separate numerical and categorical columns

numerical\_cols = df.select\_dtypes(include=['int64', 'float64']).columns

categorical\_cols = df.select\_dtypes(include=['object', 'category']).columns

# Preprocessing for numerical data: Impute missing values and normalize

numerical\_transformer = Pipeline(steps=[

('imputer', SimpleImputer(strategy='mean')),

('scaler', StandardScaler())

])

# Preprocessing for categorical data: Impute missing values and one-hot encode

categorical\_transformer = Pipeline(steps=[

('imputer', SimpleImputer(strategy='most\_frequent')),

('onehot', OneHotEncoder(handle\_unknown='ignore'))

])

# Combine transformations using ColumnTransformer

preprocessor = ColumnTransformer(

transformers=[

('num', numerical\_transformer, numerical\_cols),

('cat', categorical\_transformer, categorical\_cols)

]

)

# Apply transformations

df\_processed = preprocessor.fit\_transform(df)

# Get the feature names after encoding

num\_features = numerical\_cols

cat\_features = preprocessor.named\_transformers\_['cat']['onehot'].get\_feature\_names\_out(categorical\_cols)

feature\_names = list(num\_features) + list(cat\_features)

# Convert to DataFrame

df\_processed = pd.DataFrame(df\_processed, columns=feature\_names)

return df\_processed

# Example usage:

data = {

'age': [25, 30, 35, 40, None],

'income': [50000, 60000, 70000, 80000, 90000],

'gender': ['male', 'female', None, 'female', 'male'],

'city': ['New York', 'Los Angeles', 'Chicago', None, 'New York']

}

df = pd.DataFrame(data)

cleaned\_df = preprocess\_data(df)

print(cleaned\_df)

1. **Write a Python function to compute the nth Fibonacci number using recursion.**

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)

# Example usage:

n = 10

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

1. **Write a Python function that divides two numbers and handles the case where the divisor is zero by returning a custom error message.**

def divide\_numbers(dividend, divisor):

try:

result = dividend / divisor

except ZeroDivisionError:

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

else:

return result

# Example usage:

print(divide\_numbers(10, 2)) # Output: 5.0

print(divide\_numbers(5, 0)) # Output: Error: Division by zero is not allowed.

1. **Write a Python decorator that measures the execution time of a function and logs it. Apply this decorator to a function that performs a computationally expensive task.**

import time

import functools

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

return result

return wrapper

# Example: Function that performs a computationally expensive task

@measure\_execution\_time

def compute\_factorial(n):

factorial = 1

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

factorial \*= i

return factorial

# Example usage:

number = 1000

result = compute\_factorial(number)

print(f"{number}! = {result}")

1. **Write a Python function that takes two numbers and an operator (as a string) and performs the corresponding arithmetic operation (addition, subtraction, multiplication, or division).**

def arithmetic\_operation(num1, num2, operator):

if operator == '+':

result = num1 + num2

elif operator == '-':

result = num1 - num2

elif operator == '\*':

result = num1 \* num2

elif operator == '/':

if num2 == 0:

raise ValueError("Division by zero is not allowed")

result = num1 / num2

else:

raise ValueError("Invalid operator. Use '+', '-', '\*', or '/'.")

return result

# Example usage:

try:

num1 = 10

num2 = 5

operator = '+'

result = arithmetic\_operation(num1, num2, operator)

print(f"{num1} {operator} {num2} = {result}")

operator = '\*'

result = arithmetic\_operation(num1, num2, operator)

print(f"{num1} {operator} {num2} = {result}")

num1 = 15

num2 = 0

operator = '/'

result = arithmetic\_operation(num1, num2, operator)

print(f"{num1} {operator} {num2} = {result}")

num1 = 8

num2 = 0

operator = '/' # This will raise a ValueError

result = arithmetic\_operation(num1, num2, operator)

print(f"{num1} {operator} {num2} = {result}")

except ValueError as ve:

print(ve)

1. **Write a Python function that generates a random password. The password should contain a mix of uppercase letters, lowercase letters, digits, and special characters.**

import random

import string

def generate\_random\_password(length=12):

# Define the character sets

lowercase\_letters = string.ascii\_lowercase

uppercase\_letters = string.ascii\_uppercase

digits = string.digits

special\_characters = string.punctuation

# Combine all character sets

all\_characters = lowercase\_letters + uppercase\_letters + digits + special\_characters

# Generate a random password

password = ''.join(random.choice(all\_characters) for \_ in range(length))

return password

# Example usage:

random\_password = generate\_random\_password()

print(f"Generated Random Password: {random\_password}")

1. **Write a Python function that takes a 2D list (matrix) and returns its transpose.**

def transpose\_matrix(matrix):

if not matrix:

return []

num\_rows = len(matrix)

num\_cols = len(matrix[0])

# Create a new matrix to store the transpose

transposed\_matrix = [[0] \* num\_rows for \_ in range(num\_cols)]

# Fill the transpose matrix

for i in range(num\_rows):

for j in range(num\_cols):

transposed\_matrix[j][i] = matrix[i][j]

return transposed\_matrix

# Example usage:

matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

transposed = transpose\_matrix(matrix)

for row in transposed:

print(row)