Problem 1: Inheritance Scenario: A company has two types of employees: PermanentEmployee and ContractEmployee. Both employee types have some shared details, such as name and id, but differ in how their salaries are calculated. Permanent employees have a fixed monthly salary, while contr act employees are paid based on the number of hours worked.

Question: Define a base class Employee with attributes name and id. Create two derived classes, PermanentEmployee and ContractEmployee, that inherit from Employee. The PermanentEmployee class should have a method calculate_salary() that returns a fixed monthly salary, while ContractEmployee should have a method calculate_salary() that calculates salary based on hourly rate and hours worked. Demonstrate this setup with appropriate test data.

```
class Employee:
    def __init__(self, name, emp_id):
        self.name = name
        self.emp_id = emp_id


class PermanentEmployee(Employee):
    def __init__(self, name, emp_id, monthly_salary):
        super().__init__(name, emp_id)
        self.monthly_salary = monthly_salary

    def calculate_salary(self):
        return self.monthly_salary


class ContractEmployee(Employee):
    def __init__(self, name, emp_id, hourly_rate, hours_worked):
        super().__init__(name, emp_id)
        self.hourly_rate = hourly_rate
        self.hours_worked = hours_worked

    def calculate_salary(self):
```

```
        return self.hourly_rate * self.hours_worked


permanent_emp = PermanentEmployee("Hafiz", 101, 5000)

print(f"Permanent Employee Name: {permanent_emp.name} ID: {permanent_emp.emp_id} Salary: {permanent_emp.calculate_salary()}$")

contract_emp = ContractEmployee("Mridul", 102, 20, 120)

print(f"Contract Employee Name: {contract_emp.name} ID: {contract_emp.emp_id} Salary: {contract_emp.calculate_salary()}$")
```

Output:

```
Permanent Employee Name: Hafiz ID: 101 Salary: 5000$
Contract Employee Name: Mridul ID: 102 Salary: 2400$
```

2. Polymorphism:

Scenario: A delivery company uses different modes of transportation for various packages, such as Truck, Ship, and Plane. Each transport type has a different way of calculating the delivery cost, which includes parameters like weight and distance.

Question: Define a base class Transport with a method calculate_cost(weight, distance). Create three derived classes Truck, Ship, and Plane, each implementing calculate_cost() differently based on the transport mode. Demonstrate polymorphism by writing a function that calculates delivery costs for each mode of transportation using a single list of transport objects.

Code:

```
class Transport:
    weight = ""
    distance = ""
    def __init__(self, weight, distance):
        self.weight = weight
        self.distance = distance
    def calculate_cost(self):
```

```
        pass
class Truck(Transport):
    def calculate_cost(self):
        cost = (self.weight * 5) * (self.distance / 10)
        print("Truck Cost: ", cost)
class Ship(Transport):
    def calculate_cost(self):
        cost = (self.weight * 3) * (self.distance / 10)
        print("Ship Cost: ", cost)
class Plane(Transport):
    def calculate_cost(self):
        cost = (self.weight * 10) * (self.distance / 4)
        print("Plane Cost: ", cost)

truck = Truck(10, 20)
truck.calculate_cost()

ship = Ship(30, 10)
ship.calculate_cost()

plane = Plane(5, 4)
plane.calculate_cost()
```

OUTPUT:

Truck Cost: 100.0

Ship Cost: 90.0

Plane Cost: 125.0

3. Built-in Exception Handling Scenario:

You are working on a function that processes a list of values by dividing each by adivisor provided by the user. If the divisor is zero, an error occurs. If the divisor is a non-integeror non-numeric type, it should also raise an error.

Question: Write a function divide_elements(values, divisor) that accepts a list of numbers and a divisor.Use built-in exception handling to catch a ZeroDivisionError when dividing by zero and aTypeError if the divisor is not numeric. Ensure the program provides meaningful error messages to the user and gracefully continues processing if possible.

Code:

```
try:
    values = [10, 20, 30, 0]
    divisor = int(input("Enter a number: "))
    for i in values:
        result = i / divisor
        print(result)
except ZeroDivisionError:
    print("Value divided by zero error occurred")
except ValueError:
    print("Value error occurred")
except TypeError:
    print("Type error occurred")
except ArithmeticError:
    print("Arithmetic error occurred")
except:
    print("An error occurred")
else:
    print("Everything is Fine!")
finally:
```

```
print("Division Done!")
```

OUTPUT:

Enter a number: 0

Value divided by zero error occured

Division Done!

4. Custom Exception Handling:

Scenario: You are designing a banking application that processes withdrawals from an account.Each account has a minimum balance that must be maintained. If a withdrawal would reduce the balance below this minimum, it should raise a custom exception called InsufficientFundsError.

Question: Create a custom exception InsufficientFundsError with an appropriate message. Define a class BankAccount with attributes balance and min_balance. Write a method withdraw(amount) in BankAccount that raises an InsufficientFundsError if the withdrawal amount would bring the balance below the minimum allowed. Test the class and custom exception with a sample account.

Code:

```
class InsufficientBalanceException(Exception):
    pass
class BankAccount:
    def __init__(self, balance, min_balance):
        self.min_balance = min_balance
        self.balance = balance
    def withdraw(self, amount):
        if self.balance - amount < self.min_balance:
            raise InsufficientBalanceException
```

```
        print(self.balance - amount)
try:
    b = BankAccount(1000, 100)
    b.withdraw(5800)
except InsufficientBalanceException:
    print("ERROR: Your Remaining Balance is insufficient.")
```

OUTPUT:

ERROR: Your Remaining Balance is insufficient.

5. Numpy Functions:

Scenario: You are given a dataset of students' scores in different subjects stored in a 2D NumPy array, where each row represents a student and each column represents a subject. You need to calculate the average score per student and identify the student with the highest average. **Question:** Write a Python function using NumPy to calculate the average score for each student. Then, identify the student with the highest average score and print their score. Use appropriate NumPy functions to solve this problem efficiently.

Code:

```
import numpy as np
def average_score(scores):
    average = np.mean(scores, axis=1)
average_index = np.argmax(average)
    high_avg_score = average[average_index]
    print(f"Student : {average_index + 1}, Highest Average Score : {high_avg_score}")
    return high_avg_score
scores = np.array([
    [85, 90, 78],
    [88, 92, 80],
```

```
    [90, 91, 95],
    [70, 85, 88]
])
average_score(scores)
```

OUTPUT:

Student : 3, Highest Average Score : 92.0

6. Indexing and Slicing:

Scenario: You have a 2D NumPy array representing sales data, where each row is a product, and columns represent sales over several months. You need to analyze sales for specific months and products.

Question: Given a 2D array of sales data, write code to extract:

- Sales data for the first three products.
- Sales data for all products in the last two months.
- Sales data for a specific product and month (e.g., 2nd product in the 4th month).

Demonstrate these operations on a sample 2D array.

Code:

```
import numpy as np
sales_data = np.array([
    [100, 200, 450, 550],
    [150, 250, 300, 700],
    [500, 200, 450, 400],
    [350, 300, 200, 100]
])
products = sales_data[:3, :]
print("Sales data for the first three products:\n", products)
month_product = sales_data[:, -2:]
```

```
print("\nSales data for all products in the last two months:\n", month_product)

specific_product = sales_data[1, 3]

print("\nSales data for the 2nd product in the 4th month:", specific_product)
```

OUTPUT:

Sales data for the first three products:

[[100 200 450 550]

[150 250 300 700]

[500 200 450 400]]

Sales data for all products in the last two months:

[[450 550]

[300 700]

[450 400]

[200 100]]

7. Type Casting:

Scenario: You're working with data that was initially stored as strings in a CSV file and imported into a NumPy array. Now, you need to convert specific columns to integers or floats to perform calculations.

Question: Given a 2D NumPy array with string values, write code to:

● Convert an entire column to integer type if it represents whole numbers (e.g., age data).

● Convert a column to float type if it represents decimal values (e.g., salary or price).Demonstrate the type casting on a sample NumPy array.

Code:

```
import numpy as np

data = np.array([
   ["Hafiz", "11", "45000.50"],
   ["Riddy", "22", "55000.75"],
   ["Abir", "33", "38000.45"],
   ["Mridul", "21", "49000.25"]
])

ages = data[:, 1].astype(int)

salaries = data[:, 2].astype(float)

print("After TypeCasting Ages:", ages)

print("After TypeCasting Salaries:", salaries)
```

OUTPUT:

After TypeCasting Ages: [11 22 33 21]

After TypeCasting Salaries: [45000.5 55000.75 38000.45 49000.25]

8. Copy and View:

Scenario: You have a large dataset in a NumPy array and want to perform operations on a subset without modifying the original data. You're unsure whether to use a copy or view.

Question: Given a 2D array of data, write code to:

● Create a view of a specific row.

● Create a deep copy of a specific column.

● Modify both the view and the copy, and observe which one affects the original array. Explain the difference in behavior between the copy and view based on your code.

Code:

```
import numpy as np
```

```
a = np.array([
   [1, 2, 3, 4, 5],
   [4, 5, 6, 7, 8],
   [8, 9, 0, 1, 2],
   [6, 7, 8, 9, 0]
])
print(a)
print("\nCreate a view of third row")
print(a[2,:])
print("\nCreate a copy of third column")
print(a[:,2].copy())
print("\nModify the array")
b = a[2,:]  # View of the third row
b[:] = 0  # Set all elements of the third row to 0
print(b)
print("After Modify Original array")
print(a)
print("\nCopy of third column and modify")
c = a[:,2].copy()  # Copy of the third column
c[:] = 0  # Set all elements of the copied column to 0
print(c)
print("After Modify Original Array")
print(a)
```

OUTPUT:

```
[[1 2 3 4 5]
 [4 5 6 7 8]
 [8 9 0 1 2]
```

[6 7 8 9 0]]

Create a view of third row

[8 9 0 1 2]

Create a copy of third column

[3 6 0 8]

Modify the array

[0 0 0 0 0]

After Modify Original array

[[1 2 3 4 5]

[4 5 6 7 8]

[0 0 0 0 0]

[6 7 8 9 0]]

Copy of third column and modify

[0 0 0 0]

After Modify Original Array

[[1 2 3 4 5]

[4 5 6 7 8]

[0 0 0 0 0]

[6 7 8 9 0]]

9. Shape and Reshape:

Scenario: You have a 1D array representing data from multiple sensors over a period. To perform matrix operations, you need to restructure the data into a 2D format where each row represents a sensor and each column represents a timestamp.

Question: Given a 1D NumPy array, write code to:

● Reshape it into a 2D array with an appropriate number of rows and columns.

● If the reshaping is not possible due to mismatched elements, explain the error and provide a solution by padding/truncating the data.

Demonstrate the reshaping and necessary checks on a sample array.

Code:

```
import numpy as np
def reshape_sensor_data(sensor_data, rows, cols):
    if sensor_data.size != rows * cols:
        print(f"Error: Cannot reshape array of size {sensor_data.size} into shape ({rows}, {cols}).")
        required_size = rows * cols
        if sensor_data.size < required_size:
            padded_data = np.pad(sensor_data, (0, required_size - sensor_data.size), mode='constant')
            print("Data padded with zeros.")
        else:
            padded_data = sensor_data[:required_size]
            print("Data truncated to fit.")
        reshaped_data = padded_data.reshape(rows, cols)
    else:
        reshaped_data = sensor_data.reshape(rows, cols)
    return reshaped_data
sensor_data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
rows, cols = 3, 4
reshaped_data = reshape_sensor_data(sensor_data, rows, cols)
print("\nOriginal 1D Data:", sensor_data)
print("\nReshaped 2D Data:")
print(reshaped_data)
```

OUTPUT:

Error: Cannot reshape array of size 10 into shape (3, 4).

Data padded with zeros.

Original 1D Data: [1 2 3 4 5 6 7 8 9 10]

Reshaped 2D Data:

[[1 2 3 4]

[5 6 7 8]

[9 10 0 0]]

10. Join:

Scenario: You have two arrays representing data from two different branches of a store. To analyze combined sales, you need to merge the data by joining arrays along different axes.

Question: Given two 1D NumPy arrays representing sales, write code to:

● Join the arrays horizontally to represent two branches as columns.

● Join the arrays vertically to represent two branches as additional rows. Demonstrate both types of joins and explain in which scenario each type would be useful.

Code:

```
import numpy as np

arr1 = np.array([11, 12, 13, 14, 15, 16])

arr2 = np.array([16, 17, 18, 19, 20, 21])

print(np.stack((arr1, arr2)))

print(np.stack((arr1, arr2), axis=1))
```

OUTPUT:

Horizontally Joined Arrays (as columns):

[[11 12 13 14 15 16]

[16 17 18 19 20 21]]

Vertically Joined Arrays (as rows):

[[11 16]

[12 17]

[13 18]

[14 19]

[15 20]

[16 21]]

11. np.where :

Scenario: You're analyzing temperatures recorded over several days. You want to identify days when the temperature exceeded a certain threshold and replace any low temperatures with a specified minimum value.

Question: Given a NumPy array of temperatures, write code to:

● Use np.where to find indices where the temperature exceeds a certain threshold.

● Use np.where to replace temperatures below a certain threshold with a minimum value. Demonstrate this operation with a sample temperature array.

Code:

```
import numpy as np

temperatures = np.array([17, 12, 15, 18, 19, 20, 30, 40, 42, 22, 38, 44])

high = 20

low = 10

min_value = 10

high_temp = np.where(temperatures > high)

print("Indices where temperature exceeds the threshold:", high_temp[0])

adjusted_temperatures = np.where(temperatures < low, min_value, temperatures)

print("\nAdjusted temperatures (low values replaced):", adjusted_temperatures)
```

OUTPUT:

Indices where temperature exceeds the threshold: [6 7 8 9 10 11]

Adjusted temperatures (low values replaced): [17 12 15 18 19 20 30 40 42 22 38 44]

12. Search and Sort:

Scenario: You have a dataset representing student scores in various exams stored in an array. You need to locate specific scores and arrange the data in ascending or descending order.

Question: Given a 1D NumPy array of scores, write code to:

● Search for specific scores (e.g., 75 and 90) and return their indices.

● Sort the array in ascending and descending order. Demonstrate both operations and provide examples.

Code:

```
import numpy as np

Scores = np.array([85, 70, 90, 65, 75, 88, 95])

search_score = [75, 90]

for score in search_score:
    indices = np.where(Scores == score)[0]
    if len(indices) > 0:
        print(f"Score {score} found at indices: {indices}")
    else:
        print(f"Score {score} not found")
Ascending = np.sort(Scores)

print("\nScores in Ascending Order:", Ascending)

Descending = np.sort(Scores)[::-1]

print("Scores in Descending Order:", Descending)
```

OUTPUT:

Score 75 found at indices: [4]

Score 90 found at indices: [2]

Scores in Ascending Order: [65 70 75 85 88 90 95]

Scores in Descending Order: [95 90 88 85 75 70 65]

13. Filter:

Scenario: You have an array representing product prices, and you need to filter out products that fall within a certain price range for a sale promotion.

Question: Given a 1D array of product prices, write code to:

● Use boolean indexing to filter products within a specified price range (e.g., between $20

and $50).

● Display only the filtered prices. Demonstrate the filtering with a sample price array.

Code:

```
import numpy as np

prices = np.array([15, 25, 35, 45, 55, 65, 75])

minimum_price = 20

maximum_price = 50

filter_Price = prices[(prices >= minimum_price) & (prices <= maximum_price)]

print('Prices Between the range:', filter_Price)
```

OUTPUT:

Prices Between the range: [25 35 45]

14. Flatten:

Scenario: You have a 3D array representing data collected from multiple devices over multiple days, with each device's daily data stored in a 2D matrix format. You need to process this data a single list.

Question: Given a 3D NumPy array, write code to:

● Flatten the array into a 1D array.

● Explain why flattening might be necessary and what kind of operations it enables. Demonstrate this operation with a sample 3D array.

Code:

```
import numpy as np
```

```
data = np.array([
   [[1, 2, 3], [4, 5, 6]],
   [[7, 8, 9], [10, 11, 12]]
])
flattened_data = data.flatten()
print("Original 3D Array:\n", data)
print("\nFlattened 1D Array:\n", flattened_data)
```

OUTPUT:

Original 3D Array:

[[[1 2 3]

[4 5 6]]

[[7 8 9]

[10 11 12]]]

Flattened 1D Array:

[1 2 3 4 5 6 7 8 9 10 11 12]

15. Encapsulation:

Scenario: Imagine you are developing a software application for a bank that includes a BankAccount class. The class should manage customers' account balances, allowing deposits, withdrawals, and balance checks. For security and integrity, it's important to restrict direct access to the account balance, ensuring only authorized methods can modify or view it.

Question:

1. Create a BankAccount class that includes the following:

○ A private variable for the account balance.

○ Public methods for:

■ Deposit: Adds a specified amount to the account balance.

■ Withdraw: Subtracts a specified amount from the account balance, ensuring the balance does not go below zero.

■ Check Balance: Returns the current balance without allowing direct access to the balance variable.

2. Perform the following actions:

○ Deposit money into an account.

○ Attempt a withdrawal larger than the current balance to ensure the account does not go negative. ○ Check the balance.

Code:

```
class BankAccount:
    def __init__(self, initial_balance=0):
        self.__balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited ${amount:.2f}.")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount > self.__balance:
            print("Insufficient funds! Withdrawal denied.")
        elif amount > 0:
            self.__balance -= amount
            print(f"Withdrew ${amount:.2f}.")
        else:
            print("Withdrawal amount must be positive.")
```

```
    def check_balance(self):
        return self.__balance
if __name__ == "__main__":
    account = BankAccount(initial_balance=200)
    account.deposit(500)
    account.withdraw(400)
    account.withdraw(40)
    balance = account.check_balance()
    print(f"Current balance: ${balance:.2f}")
```

OUTPUT:

Deposited $500.00.

Withdrew $400.00.

Withdrew $40.00.

Current balance: $260.00

16. Encapsulation:

Scenario: You are working on a library management system and need to create a LibraryBook class to manage details about each book in the library. To prevent unauthorized changes to critical information, you should use encapsulation principles, specifically by employing private and protected attributes and methods.

Code:

```
class LibraryBook:
    def __init__(self, isbn, title, author):
        self.__isbn = isbn  # private attribute
        self._title = title  # protected attribute
```

```
        self._author = author  # protected attribute
        self.status = "available"  # public attribute
    def get_ISBN(self):
        return "****-****-" + self.__isbn[-4:]  # Masked ISBN

    def borrow_book(self, borrower_name):
        if self.status == "available":
            self.status = "borrowed"
            print(f"The book '{self._title}' has been borrowed by {borrower_name}.")
        else:
            print(f"The book '{self._title}' is already borrowed.")

    def _display_basic_info(self):
        print(f"Title: {self._title}")
        print(f"Author: {self._author}")

class DigitalLibraryBook(LibraryBook):
    def __init__(self, isbn, title, author, file_format):
        super().__init__(isbn, title, author)  # Call parent constructor
        self.file_format = file_format  # Specific to digital books

    def display_digital_info(self):
        self._display_basic_info()  # Call the method from LibraryBook
        print(f"File Format: {self.file_format}")  # Additional info for digital books

if __name__ == "__main__":
    book = LibraryBook("1234567890123", "The Great Gatsby", "F. Scott Fitzgerald")
```

```
print("Masked ISBN:", book.get_ISBN())
book.borrow_book("Alice")  # Borrowing the book
book.borrow_book("Bob")  # Trying to borrow again
print("\n--- Digital Library Book ---")
digital_book = DigitalLibraryBook("9876543210987", "1984", "George Orwell", "PDF")
digital_book.display_digital_info()  # Displaying digital book information
```

OUTPUT:

Masked ISBN: **-****-0123**

The book 'The Great Gatsby' has been borrowed by Alice.

The book 'The Great Gatsby' is already borrowed.

--- Digital Library Book ---

Title: 1984

Author: George Orwell

File Format: PDF