

Lab Report

Course Code:

CSE222

Course Title:

Object Oriented Programming II Lab

Topic:

Submitted To:

Teacher's Name:

Ms. Nasima Islam Bithi

Designation:

Lecturer

Department:

CSE

Daffodil International University

Submitted By:

Student Name:

Rakib Hassan

ID:

0242220005101554

Section:

63

\_M

2

Department:

CSE

Daffodil International University

Date of submission:

2025-01-04

**1. Inheritance**

class Employee: def \_\_init\_\_(self, name, id):

self.name = name

self.id = id def display(self):

return f"Employee ID: {self.id}, Name: {self.name}"

class PermanentEmployee(Employee): def \_\_init\_\_(self, name, id, monthly\_salary):

super().\_\_init\_\_(name, id) self.monthly\_salary = monthly\_salary def calculate\_salary(self): return self.monthly\_salary

class ContractEmployee(Employee): def \_\_init\_\_(self, name, id, hourly\_rate, hours\_worked):

super().\_\_init\_\_(name, id) self.hourly\_rate = hourly\_rate self.hours\_worked = hours\_worked def calculate\_salary(self):

return self.hourly\_rate \* self.hours\_worked

perm\_emp = PermanentEmployee("Mahtab", 1608, 40000) print(perm\_emp.display())

print(f"Permanent Employee Salary: ${perm\_emp.calculate\_salary()}") print()

contract\_emp = ContractEmployee("Sadman", 1951, 15, 200) print(contract\_emp.display())

print(f"Contract Employee Salary: ${contract\_emp.calculate\_salary()}") Employee ID: 1608, Name: Mahtab

Permanent Employee Salary: $40000

Employee ID: 1951, Name: Sadman

Contract Employee Salary: $3000

**2. Polymorphism**

class Transport: def calculate\_cost(self, weight, distance):

raise NotImplementedError("This method should be overridden by subclasses") class Truck(Transport): def calculate\_cost(self, weight, distance): return (weight \* 0.5 + distance \* 1.2) class Ship(Transport): def calculate\_cost(self, weight, distance): return (weight \* 0.3 + distance \* 0.8) class Plane(Transport): def calculate\_cost(self, weight, distance): return (weight \* 0.8 + distance \* 1.5) def calculate\_delivery\_costs(transports, weight, distance):

for transport in transports:

cost = transport.calculate\_cost(weight, distance) print(f"{transport.\_\_class\_\_.\_\_name\_\_} delivery cost: ${cost:.2f}") transports = [Truck(), Ship(), Plane()] weight = 67 distance = 200 print()

calculate\_delivery\_costs(transports, weight, distance) print()

Truck delivery cost: $273.50

Ship delivery cost: $180.10

Plane delivery cost: $353.60

**3. Built-in Exception Handling**

def divide\_elements(values, divisor): if not isinstance(divisor, (int, float)):

raise TypeError("The divisor must be a numeric value (int or float).") results = [] for value in values: try:

result = value / divisor results.append(result) except ZeroDivisionError: print(f"Error: Cannot divide {value} by zero.") results.append(None) except Exception as e: print(f"Unexpected error when processing {value}: {e}") results.append(None) return results values = [10, 20, 30, 0, 40] divisor = 0 try:

results = divide\_elements(values, divisor) print("Results:", results) except TypeError as te:

print(te)

Error: Cannot divide 10 by zero.

Error: Cannot divide 20 by zero.

Error: Cannot divide 30 by zero.

Error: Cannot divide 0 by zero.

Error: Cannot divide 40 by zero.

Results: [None, None, None, None, None]

**4. Custom Exception Handling**

class InsufficientFundsError(Exception): def \_\_init\_\_(self, message): super().\_\_init\_\_(message) class BankAccount: def \_\_init\_\_(self, balance, min\_balance):

self.balance = balance self.min\_balance = min\_balance def withdraw(self, amount): if self.balance - amount < self.min\_balance: print()

raise InsufficientFundsError(f"Withdrawal of {amount} exceeds the minimum balance requirement. " f"Current balance: {self.balance}, Minimum balance: {self.min\_balance}.") self.balance -= amount print()

print(f"Withdrawal successful! New balance: {self.balance}")

# Testing the class and custom exception if \_\_name\_\_ == "\_\_main\_\_":

account = BankAccount(balance=1000, min\_balance=200)

try:

account.withdraw(900) except InsufficientFundsError as e:

print(e) try:

account.withdraw(300) except InsufficientFundsError as e:

print(e) try:

account.withdraw(200) except InsufficientFundsError as e:

print(e) print()

print(f"Final balance: {account.balance}") print()

**Output:**

|  |
| --- |
| Withdrawal of 900 exceeds the minimum balance requirement. Current balance: 1000, Minimum balance: 200.  Withdrawal successful! New balance: 700 Withdrawal successful! New balance: 500  Final balance: 500 |

**5. Numpy Functions**

import numpy as np def highest\_average\_student(scores): average\_scores = np.mean(scores, axis=1) highest\_average\_index = np.argmax(average\_scores) highest\_average\_score = average\_scores[highest\_average\_index] print()

print(f"The student with the highest average score is Student {highest\_average\_index + 1} with an average score of {highest\_average\_score:.2f}") print() if \_\_name\_\_ == "\_\_main\_\_":

student\_scores = np.array([[85, 90, 78],

[92, 88, 95],

[70, 75, 80], [88, 92, 85]])

highest\_average\_student(student\_scores)

**Output:**

The student with the highest average score is sudent 2 with an average score of 91.67

**6. Indexing and Slicing** import numpy as np sales\_data = np.array([

[100, 150, 200, 250, 300, 350],

[120, 160, 220, 270, 320, 370],

[130, 180, 230, 280, 330, 380],

[140, 190, 240, 290, 340, 390],

[150, 200, 250, 300, 350, 400]

])

first\_three\_products = sales\_data[:3] print("Sales data for the first three products:") print(first\_three\_products) last\_two\_months = sales\_data[:, -2:]

print("\nSales data for all products in the last two months:") print(last\_two\_months)

specific\_product\_month = sales\_data[1, 3] print("\nSales data for the 2nd product in the 4th month:") print(specific\_product\_month) print()

**Output:**

Sales data for the first three products:

[[100 150 200 250 300 350]

[120 160 220 270 320 370]

[130 180 230 280 330 380]]

Sales data for all products in the last two months:

[[300 350]

[320 370]

[330 380]

[340 390]

[350 400]]

Sales data for the 2nd product in the 4th month: 270

**7. Type Casting** import numpy as np data = np.array([

["Ali", "25", "50000.50"],

["Maria", "30", "60000.00"],

["Sohi", "35", "70000.75"], ["Sithi", "40", "80000.00"]]) print("Original data:") print(data)

print("\nData types before conversion:") print(data.dtype)

converted\_data = np.empty(data.shape, dtype=object) converted\_data[:, 0] = data[:, 0] converted\_data[:, 1] = data[:, 1].astype(int) converted\_data[:, 2] = data[:, 2].astype(float) print("\nConverted data:")

print(converted\_data) print("\nData types after conversion:") print(converted\_data.dtype) print()

**Output:**

Original data:

[['Ali' '25' \*50000.50']

['Maria' '30' \*60000.00']

['Sohi' '35' \*70000.75']

['Sithi' '40' \*80000.00']] Data types before conversion: <U8 Converted data:

[[Ali' 25 50000.5]

[Maria 30 60000.0]

['Sohi' 35 70000.75]

['Sithi' 40 80000.0]]

Data types after conversion: object

**8. Copy and View**

import numpy as np data = np.array([

[1, 2, 3],

[4, 5, 6], [7, 8, 9],

[10, 11, 12]

])

print("Original array:") print(data) row\_view = data[1] print("\nView of the second row:") print(row\_view) column\_copy = data[:, 0].copy() print("\nDeep copy of the first column:") print(column\_copy) row\_view[0] = 100 print("\nModified view of the second row:") print(row\_view) column\_copy[0] = 200 print("\nModified copy of the first column:") print(column\_copy) print("\nOriginal array after modifications:") print(data)

**Output:**

|  |
| --- |
| Original array:  [[1 2 3]  [4 5 6] [7 8 9]  [10 11 12]]  View of the second row:  [4 5 6]  Deep copy of the first column: [1 4 7 10]  Modified view of the second row: [100 5 6] Modified copy of the first column: [200 47 10] Original array after modifications:  [[1 2 3]  [100 5 6]  [7 8 9]  [10 11 12]] |

**The difference in behavior between the copy and view based on code:**

* **View**: A view is a reference to the original data. Modifications to the view will affect the original array because they share the same memory.
* **Copy**: A deep copy creates a new array with its own memory. Modifications to the copy do not affect the original array, as they are separate entities.

This behavior is crucial for managing memory and ensuring data integrity when working with subsets of large datasets in NumPy.

**9. Shape and Reshape**

import numpy as np

data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9]) num\_sensors = 3 num\_timestamps = 4

required\_elements = num\_sensors \* num\_timestamps

# Check if reshaping is possible if data.size < required\_elements:

# If not enough elements, we need to pad the array padding\_needed = required\_elements - data.size

padded\_data = np.pad(data, (0, padding\_needed), mode='constant', constant\_values=0) reshaped\_array = padded\_data.reshape(num\_sensors, num\_timestamps)

print()

print("Original data (padded):", padded\_data) elif data.size > required\_elements:

# If too many elements, we need to truncate the array truncated\_data = data[:required\_elements]

reshaped\_array = truncated\_data.reshape(num\_sensors, num\_timestamps) print("Original data (truncated):", truncated\_data) else:

# If the sizes match, we can reshape directly

reshaped\_array = data.reshape(num\_sensors, num\_timestamps) print("Original data:", data)

# Output the reshaped array

print("Reshaped 2D array:") print(reshaped\_array) print()

**Output:**

Original data (padded): [1 2 3 4 5 6 7 89000] Reshaped 2D array:

[[1234]

[5 6 7 8]

[9000]]

1. **Join** import numpy as np a = np.array([100, 200, 150, 300, 250]) b = np.array([120, 180, 160, 290, 270]) horizontal\_join = np.column\_stack((a, b)) print("Horizontal Join (as columns):") print(horizontal\_join) vertical\_join = np.vstack((a, b)) print("\nVertical Join (as additional rows):") print(vertical\_join)

**Output:**

|  |
| --- |
| Horizontal Join (as columns):  [[100 120]  [200 180]  [150 160]  [300 290]  [250 270]]  Vertical Join (as additional rows) |
| [[100 200 150 300 250] |
| [120 180 160 290 270]] |

1. **np.where**

import numpy as np

temperatures = np.array([15, 22, 18, 30, 25, 10, 5, 28, 20, 35]) high\_temp\_threshold = 25

high\_temp\_indices = np.where(temperatures > high\_temp\_threshold)[0]

print()

print("Indices where temperature exceeds the threshold of 25°C:", high\_temp\_indices) min\_temp\_value = 10

updated\_temperatures = np.where(temperatures < 15, min\_temp\_value, temperatures) print("\nUpdated temperatures:") print(updated\_temperatures) **Output:**

Indices where temperature exceeds the threshold of 25°C: [379]Updated temperatures:

[15 22 18 30 25 10 10 28 20 35]

**12. Search and Sort** import numpy as np

scores = np.array([85, 70, 90, 75, 60, 95, 80, 100, 55, 88]) search\_scores = [75, 90]

indices = {score: np.where(scores == score)[0] for score in search\_scores} print("Indices of specific scores:") for score, idx in indices.items():

if idx.size > 0:

print(f"Score {score} found at indices: {idx}") else:

print(f"Score {score} not found.") sorted\_ascending = np.sort(scores) print("\nScores sorted in ascending order:") print(sorted\_ascending)

sorted\_descending = np.sort(scores)[::-1] print("\nScores sorted in descending order:") print(sorted\_descending)

**Output:**

Indices of specific scores:

Score 75 found at indices: [3]

Score 90 found at indices: [2]

Scores sorted in ascending order: [55 60 70 75 80 85 88 90 95 100] Scores sorted in descending order:

[100 95 90 88 85 80 75 70 60 55]

1. **Filter** import numpy as np

product\_prices = np.array([15.99, 22.50, 30.00, 45.75, 50.00, 60.00, 75.25, 10.00, 40.00]) lower\_bound = 20.00 upper\_bound = 50.00

filtered\_prices = product\_prices[(product\_prices >= lower\_bound) & (product\_prices <= upper\_bound)] print("Filtered product prices within the range $20 to $50:") print(filtered\_prices)

**Output:**

PS C:105er's (acid1 \Desktop\ру стоп C./Program FIEST

Filtered product prices within the range $20 to $50: [22.5 30. 45.75 50. 40. ]

1. **Flatten** import numpy as np data = np.array([

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

[[10, 11, 12], [13, 14, 15], [16, 17, 18]]

])

flattened\_data = data.flatten() print("Flattened data:") print(flattened\_data)

**Output:**

Flattened data:

[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18]

**15. Encapsulation**

class BankAccount: def \_\_init\_\_(self): self.\_\_balance = 0.0 def deposit(self, amount): if amount > 0:

self.\_\_balance += amount

print(f"Deposited: ${amount:.2f}.\n New balance: ${self.\_\_balance:.2f}.") else:

print("Deposit amount must be positive.")

def withdraw(self, amount): if amount > self.\_\_balance: print("Insufficient funds.") elif amount <= 0:

print("Withdrawal amount must be positive.") else:

self.\_\_balance -= amount

print(f"Withdrew: ${amount:.2f}. New balance: ${self.\_\_balance:.2f}.")

def check\_balance(self): return f"Current balance: ${self.\_\_balance:.2f}" account = BankAccount() account.deposit(100.00) account.withdraw(150.00) print(account.check\_balance())

**Output:**

Deposited: $100.00.

New balance: $100.00.

Insufficient funds.

Current balance: $100.00

**16. Encapsulation**

class LibraryBook: def \_\_init\_\_(self, title, author, isbn):

self.\_\_isbn = isbn self.\_title = title self.\_author = author self.\_status = "available" def get\_ISBN(self):

return f"ISBN: {self.\_\_isbn[:3]}-\*\*\*\*-\*\*\*\*-{self.\_\_isbn[-3:]}" def borrow\_book(self, borrower\_name):

if self.\_status == "available": self.\_status = "borrowed"

print(f"{self.\_title} has been borrowed by {borrower\_name}.") else:

print(f"{self.\_title} is currently not available for borrowing.")

def \_display\_basic\_info(self):

print(f"Title: {self.\_title}, Author: {self.\_author}") class DigitalLibraryBook(LibraryBook): def \_\_init\_\_(self, title, author, isbn, file\_format):

super().\_\_init\_\_(title, author, isbn) self.file\_format = file\_format def display\_info(self): self.\_display\_basic\_info() print(f"File Format: {self.file\_format}")

book = LibraryBook("The Great Gatsby", "F. Scott Fitzgerald", "9780743273565") print(book.get\_ISBN()) book.borrow\_book("Alice")

digital\_book = DigitalLibraryBook("Digital Fortress", "Dan Brown", "9780440240823", "PDF") digital\_book.display\_info()

**Output:**

PS C.\USERS (AcM01 \DESKTOP\ру сполу

ISBN: 978---565

The Great Gatsby has been borrowed by Alice.

Title: Digital Fortress, Author: Dan Brown

File Format: PDF