1.1 Provide the Python code to create a dictionary with at least four key-value pairs. Demonstrate how to access a second element using its key and update its value. [5]

#Create a dictionary with four key-value pairs

Students\_dict = {

1: 'John',

2: 'Rodney', [2]

3: 'Keshiv',

4: 'Aveshin'

}

#Access the second element using its key

print(students\_dict[2]) [1]

#Update the value of the second element students\_dict[2] = 'Sisanda' [2]

(print(students\_dict[2])

1.3 Using a Python code example, demonstrate polymorphism. [5] Polymorphism allows for flexibility and dynamic usage, enabling a single interface or method name to be used with different data types or objects (many forms)

class Animal: def sound(self):

pass

class Dog(Animal): def sound(self): return "Bark!"

class Cat(Animal): def sound(self): return "Meow!"

#W e have a base class **Animal** with a method **sound()**. We then have two derived classes **Dog** and **Cat** that override the **sound()** method.

def animal\_sound(animal): return animal.sound()

dog = Dog() cat = Cat()

print(animal\_sound(dog)) print(animal\_sound(cat))

2.4 Define a class named Employee with attributes: empnumber, name, surname, and age. Include methods to set and get each attribute.[5]

class Employee: def \_\_init\_\_(self, empnumber, name, surname, age):

self.\_empnumber = empnumber self.\_name = name self.\_surname = surname self.\_age = age

# Getter and Setter for empnumber def get\_empnumber(self): return self.\_empnumber

def set\_empnumber(self, empnumber):

self.\_empnumber = empnumber

# Getter and Setter for name def get\_name(self): return self.\_name

def set\_name(self, name):

self.\_name = name

# Getter and Setter for surname def get\_surname(self): return self.\_surname

def set\_surname(self, surname):

self.\_surname = surname

# Getter and Setter for age def get\_age(self): return self.\_age

def set\_age(self, age): self.\_age = age

# Example Usage emp = Employee(123, "Ivan", "Rakitic", 34) print(emp.get\_name()) emp.set\_name("Isco")

print(emp.get\_name())

NB: LECTURER DISCRETION ON THIS

2.5 Using the Employee class above, create a child class Pensioneer with an additional attribute pension\_amount that checks if the pensioneer is eligible for amount greater than R7000

[5]

class Pensioneer(Employee):

def \_\_init\_\_(self, empnumber, name, surname, age, pension\_amount):

super().\_\_init\_\_(empnumber, name, surname, age) # Call the constructor of the parent class self.\_pension\_amount = pension\_amount

#Getter and Setter for pension\_amount def get\_pension\_amount(self): return self.\_pension\_amount

def set\_pension\_amount(self, pension\_amount): self.\_pension\_amount = pension\_amount

#Method to check eligibility for pension amount greater than R7000 def is\_eligible\_for\_high\_pension(self):

return self.\_pension\_amount > 7000

pensioner1 = Pensioneer(4025, "Erling", "Haarland", 65, 7500) print(pensioner1.get\_name()) print(pensioner1.is\_eligible\_for\_high\_pension())

2.6 Using Python code examples, demonstrate the difference between object-oriented programming abstraction and encapsulation [10]

**Abstraction** is the process of hiding the complex implementation details and showing only the essential features of an object. It simplifies complex systems by breaking them down into more manageable parts and showcasing only what's necessary.

A class Car has many parts but we can abstract and define it with essential features like **start()**, **stop()**, and **drive()**

class Car: def start(self):

#implementation to start the car print("Car started")

def drive(self):

#implementation to drive the car print("Car is moving")

def stop(self):

#implementation to stop the car print("Car stopped")

**Encapsulation** involves bundling data (attributes) and methods (functions) that operate on the data into a single unit or class. It also restricts direct access to some of the object's components, which is a means of preventing unintended interference and misuse of the data.

Using the **Person class**, where we want to restrict the direct modification of the age attribute:

class Person: def \_\_init\_\_(self, name, age):

self.\_name = name # protected attribute (by convention) self.\_\_age = age # private attribute

# Getter method def get\_age(self): return self.\_\_age

# Setter method def set\_age(self, age): if age > 0:

self.\_\_age = age else:

print("Invalid age")

def display(self):

print(f"{self.\_name} is {self.\_\_age} years old")

2.7 Write a Python program that inputs the first 10 odd integers and stores them in the tuple. The program should receive its input from the user’s keyboard. (Use a While or For loop) [10]

#Initialize an empty list to store odd numbers odd\_numbers = []

#Counter to keep track of number of odd integers entered count = 0

#Loop until we have 10 odd integers while count < 10: try:

#Take input from user num = int(input(f"Enter odd integer {count + 1}: "))

#Check if the number is odd and not already in the list if num % 2 != 0 and num not in odd\_numbers:

odd\_numbers.append(num) count += 1 else: print("That's not a unique odd number. Please enter again.") except ValueError:

print("Please enter a valid integer.")

#Convert list to tuple odd\_tuple = tuple(odd\_numbers)

print("\nThe tuple of 10 odd integers is:", odd\_tuple)

2.8 Create a two-dimensional list with 5 elements, update the list with two more two-dimensional elements, and print the list in reverse [10]

#Create a two-dimensional list with 5 elements two\_dim\_list = [

[1, 2],

[3, 4],

[5, 6],

[7, 8],

[9, 10]

]

#Update the list with two more two-dimensional elements two\_dim\_list.append([11, 12]) two\_dim\_list.append([13, 14])

#Print the updated list in reverse reversed\_list = two\_dim\_list[::-1] for sublist in reversed\_list:

print(sublist)