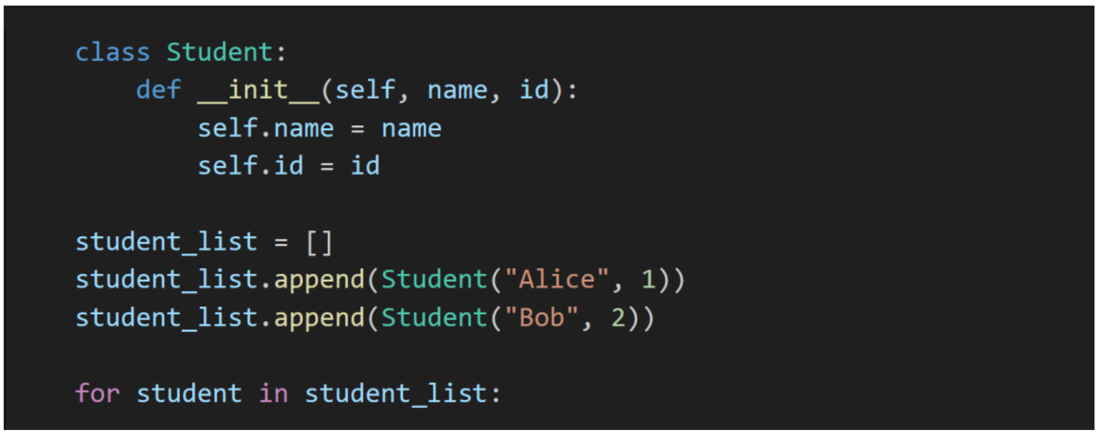
Unit 7: Debugging / Error Handling, Data Structures and Data Search

e-Portfolio Activities:

1. Discuss the ways in which data structures support object-oriented development. Use examples of three different data structures to contextualise your response.

Data structures play a crucial role in supporting object-oriented development by providing ways to organize and manipulate data objects effectively. Python offers a variety of built-in data structures that can be used in object-oriented programming to manage and store objects. Here are three different data structures in Python and how they support object-oriented development:

1. Lists: Lists are a versatile and commonly used data structure in Python that can store elements of different data types. In object-oriented development, lists can be used to store and manage collections of objects of a specific class. By using lists, you can easily iterate over objects, add or remove elements, and perform various operations on them.



```python
class Student:
    def __init__(self, name, id):
        self.name = name
        self.id = id

student_list = []
student_list.append(Student("Alice", 1))
student_list.append(Student("Bob", 2))

for student in student_list:
        print(student.name)
```

2. Dictionaries: Dictionaries are another powerful data structure in Python that stores key-value pairs. In object-oriented development, dictionaries can be used to map objects to specific keys or identifiers. This can be useful for quick lookup and retrieval of object information based on a unique key.

```python
class Student:
    def __init__(self, name, id):
        self.name = name
        self.id = id

students_dict = {}
students_dict[1] = Student("Alice", 1)
students_dict[2] = Student("Bob", 2)

print(students_dict[2].name)
```

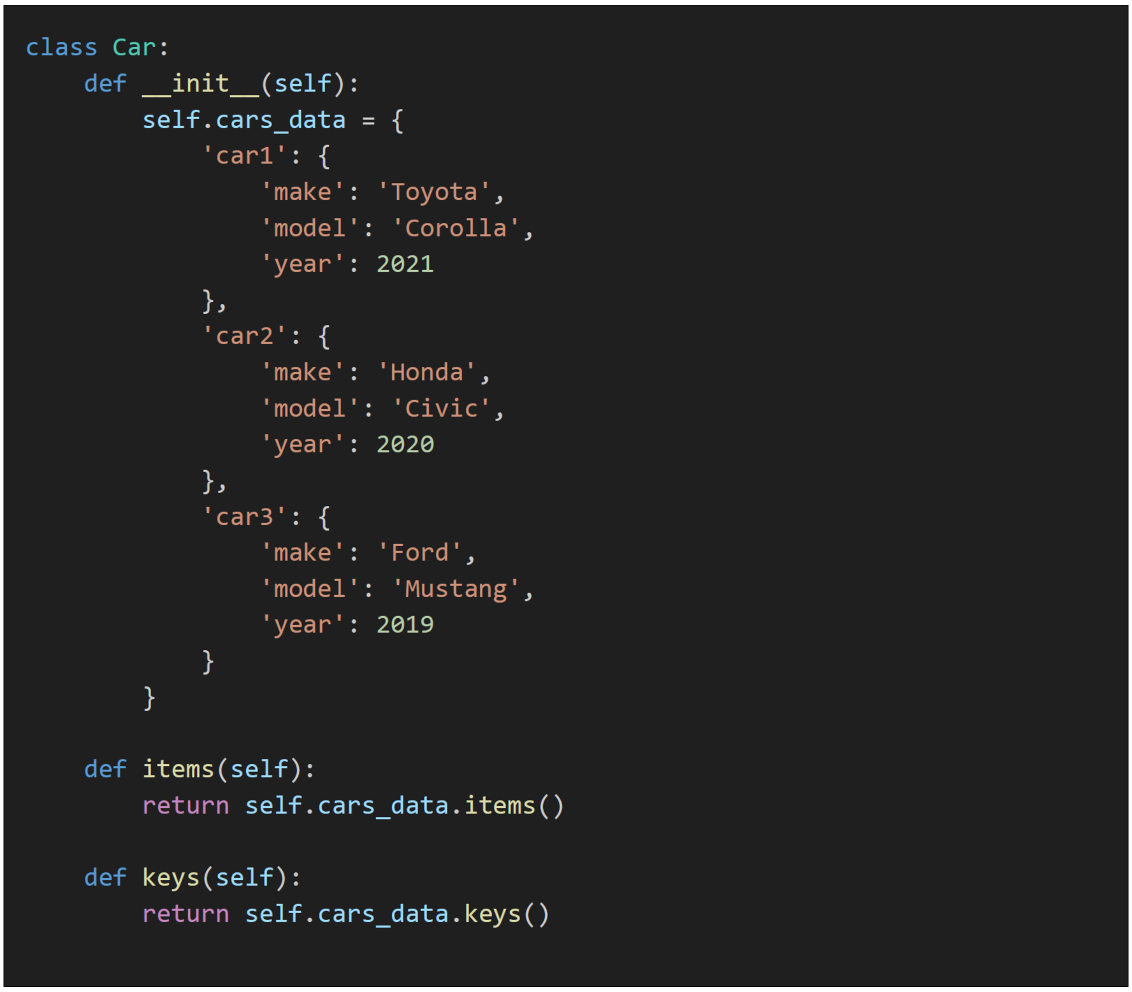
3. Sets: Sets are unordered collections of unique elements in Python. In object-oriented development, sets can be used to store and operate on unique objects within a class. Sets are particularly useful when you need to perform operations such as intersection, union, or difference on objects.

```python
class Product:
    def __init__(self, name):
        self.name = name

product_set = set()
product_set.add(Product("Laptop"))
product_set.add(Product("Mouse"))
product_set.add(Product("Laptop"))

for product in product_set:
    print(product.name)
```

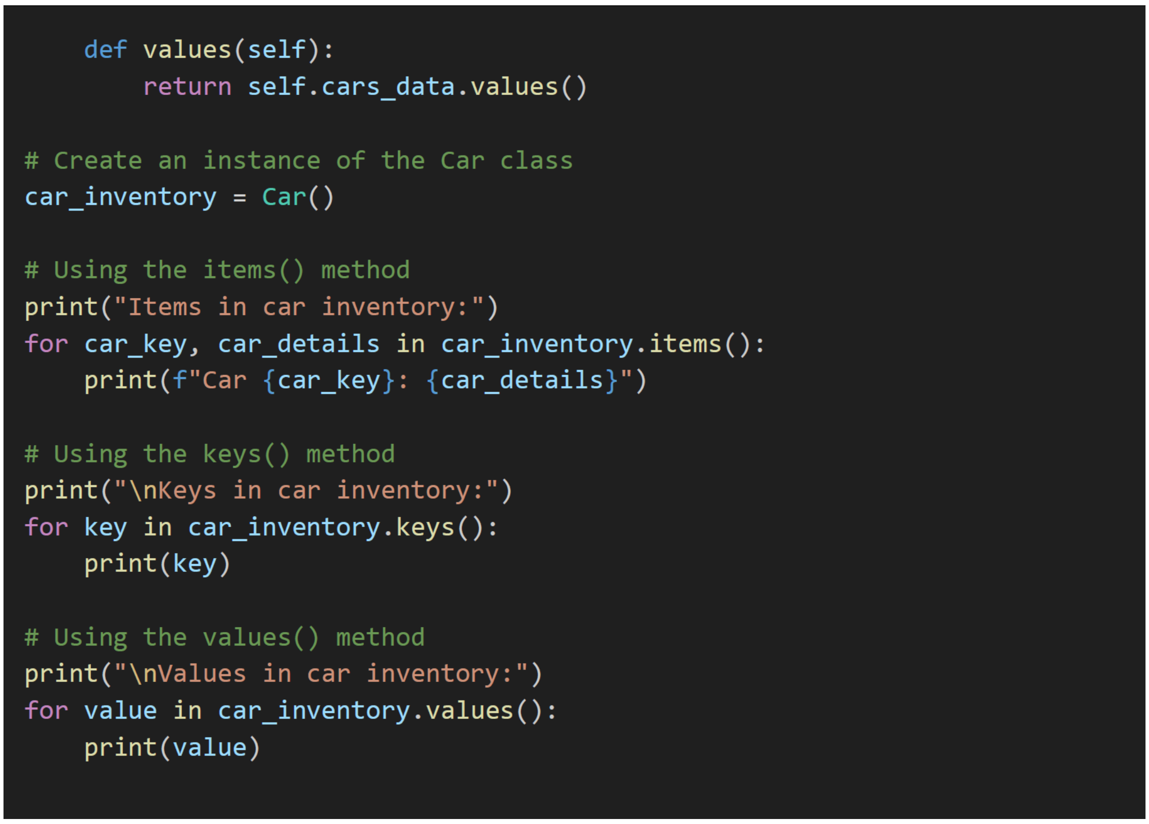
The versatility and flexibility of data structures such as lists, dictionaries, and sets play a crucial role in supporting object-oriented development. These data structures enable developers to efficiently manage and manipulate objects within a class, facilitating the implementation of object-oriented programming principles like encapsulation, inheritance, and polymorphism.

2. Create a nested dictionary of data on cars within a Car class. Extend the program to work with the dictionary by calling the following methods:

- ***items()***
- ***keys()***
- ***values()***



```python
class Car:
    def __init__(self):
        self.cars_data = {
            'car1': {
                'make': 'Toyota',
                'model': 'Corolla',
                'year': 2021
            },
            'car2': {
                'make': 'Honda',
                'model': 'Civic',
                'year': 2020
            },
            'car3': {
                'make': 'Ford',
                'model': 'Mustang',
                'year': 2019
            }
        }

    def items(self):
        return self.cars_data.items()

    def keys(self):
        return self.cars_data.keys()
```



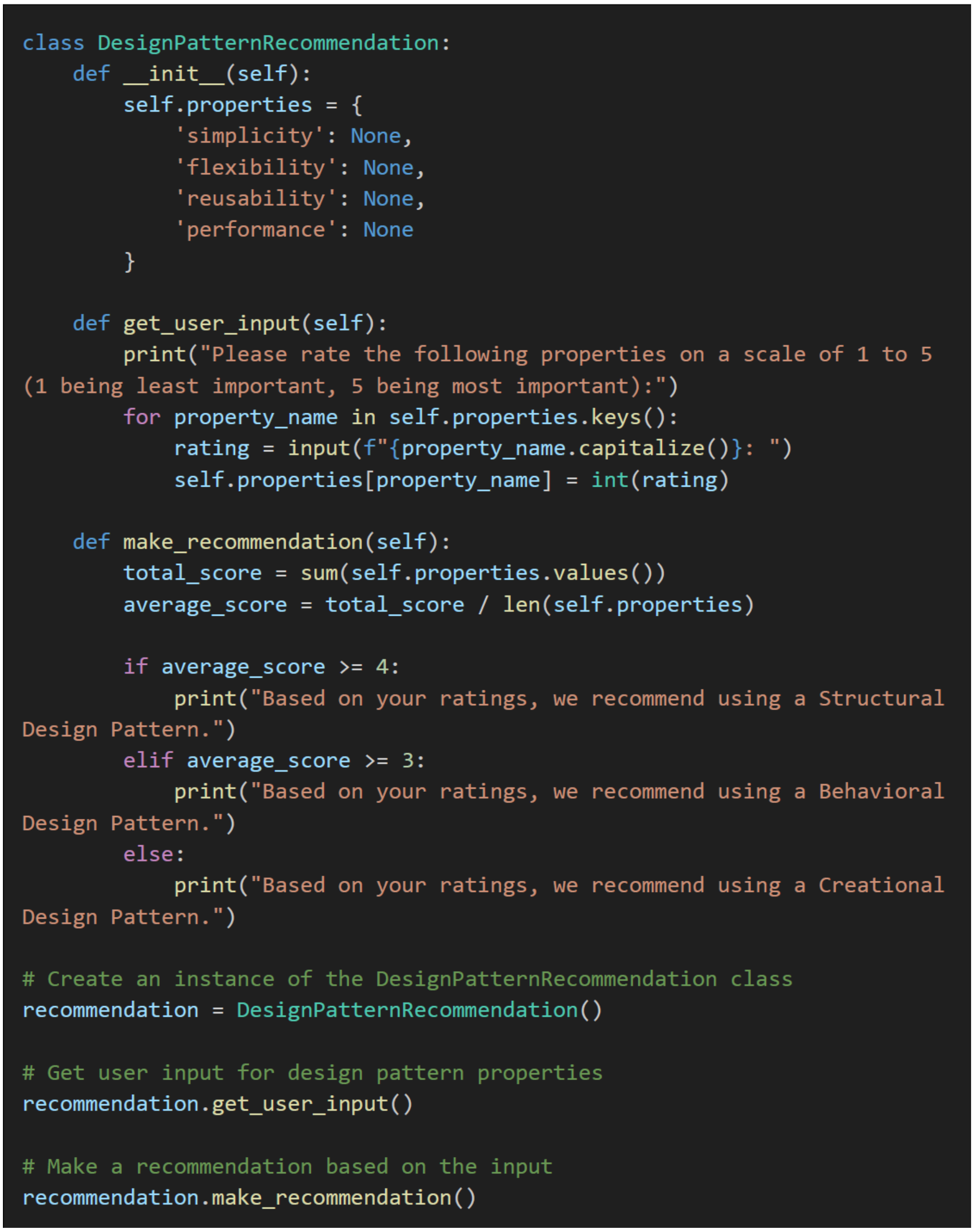
```
    def values(self):
        return self.cars_data.values()

# Create an instance of the Car class
car_inventory = Car()

# Using the items() method
print("Items in car inventory:")
for car_key, car_details in car_inventory.items():
    print(f"Car {car_key}: {car_details}")

# Using the keys() method
print("\nKeys in car inventory:")
for key in car_inventory.keys():
    print(key)

# Using the values() method
print("\nValues in car inventory:")
for value in car_inventory.values():
    print(value)
```

In the above example, it is defined a Car class with a nested dictionary cars_data containing information about different cars. The class defines methods items(), keys(), and values() to work with the dictionary data. The items() method returns a list of key-value pairs, the keys() method returns a list of keys, and the values() method returns a list of values in the nested dictionary.

When program runs, it will display the items, keys, and values in the car inventory based on the methods we implemented in the Car class.

3. Read the article by Kampffmeyer & Zschaler (2007). Develop a program which allows a user to enter the properties which they require of a design pattern, and have the program make a recommendation. Your program should use a constructor to initialise attributes and assign values to variables based on input entered by the user.

Disclaimer: article by Kampffmeyer & Zschaler (2007) is not provided as source, only in-text citation, however any source.



```python
class DesignPatternRecommendation:
    def __init__(self):
        self.properties = {
            'simplicity': None,
            'flexibility': None,
            'reusability': None,
            'performance': None
        }

    def get_user_input(self):
        print("Please rate the following properties on a scale of 1 to 5
(1 being least important, 5 being most important):")
        for property_name in self.properties.keys():
            rating = input(f"{property_name.capitalize()}: ")
            self.properties[property_name] = int(rating)

    def make_recommendation(self):
        total_score = sum(self.properties.values())
        average_score = total_score / len(self.properties)

        if average_score >= 4:
            print("Based on your ratings, we recommend using a Structural
Design Pattern.")
        elif average_score >= 3:
            print("Based on your ratings, we recommend using a Behavioral
Design Pattern.")
        else:
            print("Based on your ratings, we recommend using a Creational
Design Pattern.")

# Create an instance of the DesignPatternRecommendation class
recommendation = DesignPatternRecommendation()

# Get user input for design pattern properties
recommendation.get_user_input()

# Make a recommendation based on the input
recommendation.make_recommendation()
```

In the above program, it is defined a DesignPatternRecommendation class with attributes representing different properties of a design pattern. The class has a constructor that initializes these properties with None. The get_user_input() method prompts the user to rate the properties on a scale of 1 to 5 and assigns these ratings to the corresponding properties. The make_recommendation() method calculates the average score of the properties and makes a recommendation based on the average score.

When program runs, the user will be prompted to enter ratings for different design pattern properties, and based on the input, the program will recommend a type of design pattern to use.