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 are essential in the field of object-oriented development. They make it possible to store and organise data effectively and conveniently. As a result, they are excellent choices for illustrating item shapes and their connections (opens-server.cs.vt.edu, N.D.).

Opens-server.cs.vt.edu (N.D.) states that object-oriented programming has four fundamental principles: Encapsulation, Inheritance, Polymorphism, and Abstraction. Encapsulation creates self-contained classes to protect data privacy and define rules for visibility and modifications. Inheritance allows for code reuse through class hierarchies. Polymorphism enables creation procedures for objects with unknown types until runtime. Abstraction develops software objects to represent real-world objects and creates a model or view while hiding implementation details. These principles are crucial for efficient and effective software development.

Listed below are three distinct data structures and their roles in supporting object-oriented development:

1.1 Lists are a very flexible data structure that can hold components of different types and dynamically change their size. Lists are widely used to represent collections of objects or properties within an object in the context of object-oriented programming (Rai, 2023). As an illustration, let's look at the 'Employee' class from OneCompiler (N.D):

```
class Employee:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.skills = []  # List to store employee's skills

    def add_skill(self, skill):
        self.skills.append(skill)
```

In this case, the employee's skills are listed in the skills attribute of the Employee class. The add_skill method is used to include new skills in the list. For each employee, lists make it possible to store and manage a different number of skills, which encourages skill encapsulation in the Employee object.

1.2 A dictionary is a flexible data structure that keeps track of a group of objects. It consists of a collection of keys connected to an associated value. The dictionary searches for and returns the value corresponding to each key (en.wikibooks.org, n.d.). Consider the following example of the class 'Student':

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.grades = {}  # Dictionary to store course grades

    def add_grade(self, course, grade):
        self.grades[course] = grade
```

As an illustration, the 'Student' class contains an attribute called grades, a dictionary that connects course names with corresponding grades. New grades can be added to the dictionary by employing the add_grade method. The 'Student' object's encapsulation and grade data management are supported by dictionaries, which provide a quick way to access and retrieve grades according to course titles.

1.3 A stack is a data structure that adheres to the Last-In-First-Out (LIFO) rule. This implies that the item that was most recently introduced will be the one removed first. The stack has several components, but we can only access the top one. A pointer to the top element must be established to form a stack (Isaac Computer Science, N.D.). Let's see an illustration of a method invocation stack to provide context:

```
class MethodStack:
    def __init__(self):
        self.stack = []

    def push_method(self, method_name):
        self.stack.append(method_name)

    def pop_method(self):
        return self.stack.pop()

stack = MethodStack()
stack.push_method("method1")
stack.push_method("method2")
method = stack.pop_method()
```

```
print(method)  # Outputs: "method2"
```

This example shows how the MethodStack class, which uses a list to create stack-like capabilities, is used. While the pop_method function removes and gets the most recently added method name, the push_method function adds the method name to the stack. Stacks play a key role in the systematic execution of methods inside object-oriented systems, ensuring that the most recent invoked method is fully executed and completed before moving on to the previous one.

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(), and values()

Information can be organised hierarchically using nested dictionaries. This essential tool, a layered dictionary, is used to manage complex data. It functions similarly to "records" or "structs" in other programming languages (Learn By Example, 2019).

Based on the concepts of rowannicholls.github.io (N.D.), the following code extends the programme to work with the dictionary by invoking the items(), keys(), and values() methods. It creates a nested dictionary of data about vehicles within a Car class:

```
class Car:
    def __init__(self):
        self.car_data = {
            'Vauxhall': {
                'model': 'Mokka',
                'year': 2021,
                'colour': 'Black'
            },
            'Toyota': {
                'model': 'Aygo',
                'year': 2018,
                'colour': 'Blue'
            },
            'Ford': {
                'model': 'Fiesta',
                'year': 2020,
                'colour': 'Red'
            }
        }

    def get_car_data(self):
        return self.car_data
```

```
    def print_car_data_items(self):
        for brand, details in self.car_data.items():
            print(f"Brand: {brand}")
            for key, value in details.items():
                print(f"{key}: {value}")
            print()

    def print_car_data_keys(self):
        for brand in self.car_data.keys():
            print(brand)

    def print_car_data_values(self):
        for details in self.car_data.values():
            print(details)

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

# Access and print the nested dictionary of car data
car_data = car.get_car_data()
print(car_data)

# Print car data using the items() method
print("\nItems():")
car.print_car_data_items()

# Print car brand names using the keys() method
print("\nKeys():")
car.print_car_data_keys()

# Print car details using the values() method
print("\nValues():")
car.print_car_data_values()
```

Here is an illustration: The "car_data" nested dictionary, part of the Car class, contains information about numerous car brands. We can access this dictionary using the "get_car_data()" method. The "print_car_data_items()", "print_car_data_keys()", and "print_car_data_values()" methods also demonstrate how to utilise the "items()", "keys()", and "values()" methods to modify the dictionary.

3. **Read the article by Kampffmeyer & Zschaler (2007). Develop a program allowing users to enter the properties 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.**

According to Kampffmeyer & Zschaler (2007), software development can be daunting when finding the appropriate design pattern. However, Kampffmeyer & Zschaler (2007)

have developed a tool that provides pattern suggestions based on a problem description, utilising the Design Pattern Intent Ontology (DPIO). The DPIO formalises the intent behind the 23 patterns illustrated in the Gang of Four's book. Our paper refines Tichy's classification for computer-aided querying by utilising ontologies to classify over 100 design patterns based on their intent (Kampffmeyer & Zschaler, 2007).

They have developed a Design Pattern Wizard that, given a description of a design challenge, suggests appropriate design patterns. We can locate the pattern we require for our programme using their ontology and tool.

According to Kampffmeyer & Zschaler (2007), we must present our design challenge as a collection of problem types and problem parameters to use their ontology. Design difficulties can be categorised as problems kinds like changeability, extensibility, or efficiency. Problem parameters are more detailed features of the issue, including the number of alternatives, the number of objects, or the amount of communication required.

Here is a code based on the concepts of Kampffmeyer & Zschaler (2007) that enable users to specify their desired design pattern properties and receive suitable recommendations. The program utilizes a constructor to initialise attributes and allocate values to variables according to the user's inputs:

```
class DesignPattern:
    def __init__(self, name, intent, problem, solution):
        self.name = name
        self.intent = intent
        self.problem = problem
        self.solution = solution

    def recommend_pattern(self, required_intent, required_problem, required_solution):
        if self.intent == required_intent and self.problem == required_problem and self.solution ==
required_solution:
            return f"The {self.name} pattern is recommended based on your requirements."
        else:
```

```
        return "No matching design pattern found."

design_patterns = []

# Get user input for required design pattern properties
required_intent = input("Enter the required intent: ")
required_problem = input("Enter the required problem: ")
required_solution = input("Enter the required solution: ")

# Iterate through the design patterns and make a recommendation
for pattern in design_patterns:
    recommendation = pattern.recommend_pattern(required_intent, required_problem,
required_solution)
    if recommendation != "No matching design pattern found.":
        print(recommendation)
        break
else:
    print("No matching design pattern found.")
```

In this case, a design pattern with attributes like name, intent, problem, and solution is embodied by the DesignPattern class. The constructor for initialising these attributes is the __init__ function. The recommend_pattern method compares the attributes of the current design pattern instance with the user's desired design pattern properties (intent, problem, and solution). If a match is found, a recommendation message is given. If not, it generates a notice that reads, "No matching design pattern found." The software compiles a list of readily available design patterns and asks users to provide the needed properties. Once it finds a matching pattern, it stops iterating through the design patterns and calls the recommend_pattern method for each one. If no pattern matches, a message stating "No matching design pattern found" is printed.

References:

opendsa-server.cs.vt.edu. (N.D.). 4.2. Introduction to Object Oriented Programming — OpenDSA Data Structures and Algorithms Modules Collection. [online] Available at: https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/IntroOO.html#:~:text=Object%2Doriented%20programming%20(OOP).

Rai S. (2023). *Understanding Lists in Python: An In-Depth Overview*. [online] Available at: https://blog.saurabhraidev.co/understanding-lists-in-python.

onecompiler.com. (n.d.). *Employee.py - Python - OneCompiler*. [online] Available at: https://onecompiler.com/python/3ve9mbs3t.

en.wikibooks.org. (n.d.). *Fundamentals of data structures: Dictionaries - Wikibooks, open books for an open world*. [online] Available at: https://en.wikibooks.org/wiki/A-level_Computing/AQA/Paper_1/Fundamentals_of_data_structures/Dictionaries.

Isaac Computer Science. (n.d.). *Isaac Computer Science*. [online] Available at: https://isaaccomputerscience.org/concepts/dsa_datastruct_stack?examBoard=all&stage=all.

Learn By Example. (2019). *Python Nested Dictionary*. [online] Available at: https://www.learnbyexample.org/python-nested-dictionary/ [Accessed 19 Jul. 2023].

Holger Kampffmeyer and Steffen Zschaler (2007). Finding the Pattern You Need: The Design Pattern Intent Ontology. pp.211–225. doi:https://doi.org/10.1007/978-3-540-75209-7_15.