Assignment 1: Discuss the ways in which data structures support object-oriented development

Data structures are fundamental components in programming that allow storing and organizing data. They are crucial in object-oriented development as they enable data encapsulation within objects and enhance code readability and maintainability. Here are three different data structures that support object-oriented development:

1. List

For example, can be used when managing a collection of student objects in a class.

Lists allow us to store multiple objects in a single container. In an object-oriented program, a Student class can be defined to encapsulate student-related data. A list can then be used to store instances of the Student class, enabling easy access, modification, and iteration over the student collection.

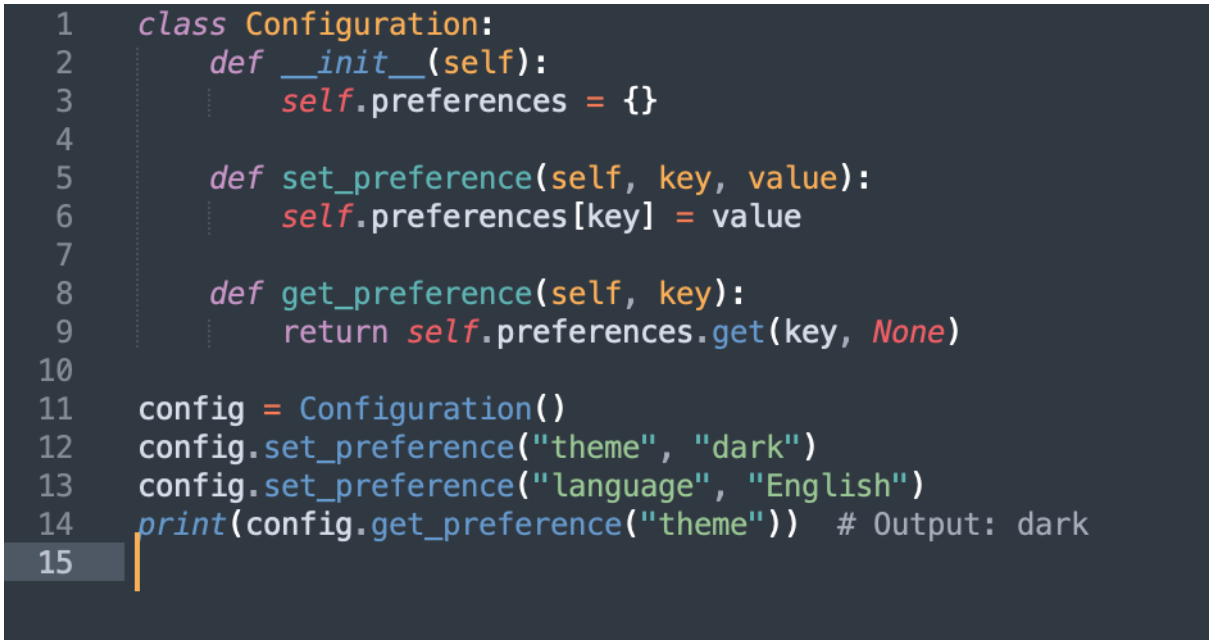
```python
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

students = [Student("Alice", 20), Student("Bob", 22), Student("Charlie", 23)]
for student in students:
    print(student.name, student.age)
```

2. Dictionary

For example, can be used for storing user preferences in a configuration object.

Dictionaries allow us to store key-value pairs, which is particularly useful for configurations and settings. In an object-oriented program, a Configuration class can

use a dictionary to store user preferences, allowing for quick access and modification of settings using keys.



```python
class Configuration:
    def __init__(self):
        self.preferences = {}

    def set_preference(self, key, value):
        self.preferences[key] = value

    def get_preference(self, key):
        return self.preferences.get(key, None)

config = Configuration()
config.set_preference("theme", "dark")
config.set_preference("language", "English")
print(config.get_preference("theme"))  # Output: dark
```

3. Set

For example, can be used for managing a collection of unique tags for blog posts.

Sets are used to store unique elements. For example, a BlogPost class can use a set to store tags, ensuring that each tag is unique and providing efficient operations for uniquely identifying users for website operations such as tracking, and content personalization.

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:

```python
class Car:
    def __init__(self):
        self.car_data = {
            "Toyota": {
                "model": "Corolla",
                "year": 2020,
                "features": ["Air Conditioning", "Bluetooth", "Cruise Control"]
            },
            "Ford": {
                "model": "Mustang",
                "year": 2019,
                "features": ["Leather Seats", "Navigation System", "Backup Camera"]
            },
            "Tesla": {
                "model": "Model 3",
                "year": 2021,
                "features": ["Autopilot", "Electric", "Panoramic Roof"]
            }
        }

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

    def keys(self):
        return self.car_data.keys()

    def values(self):
        return self.car_data.values()

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

# Call the methods and print their outputs
print("Items:", car_instance.items())
print("Keys:", car_instance.keys())
print("Values:", car_instance.values())
```

3. 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

Steps:

1) Defines a DesignPatternRecommender class to recommend design patterns based on user input.
2) Uses a constructor to initialize the class with a dictionary of design patterns.
3) The recommend_pattern method matches user input to recommend a suitable design pattern.

```python
class DesignPatternRecommender:
    def __init__(self):
        self.patterns = {
            "Singleton": {
                "purpose": "Ensure a class has only one instance",
                "example": "Logging",
                "complexity": "Low"
            },
            "Factory": {
                "purpose": "Create objects without specifying the exact class",
                "example": "GUI Components",
                "complexity": "Medium"
            },
            "Observer": {
                "purpose": "Notify dependent objects of state changes",
                "example": "Event Handling",
                "complexity": "High"
            }
        }

    def recommend_pattern(self, purpose=None, complexity=None):
        for name, details in self.patterns.items():
            if (purpose and purpose.lower() in details["purpose"].lower()) or (complexity and complexity.lower() in details["complexity"].lower()):
                return f"Recommended Pattern: {name}, Purpose: {details['purpose']}, Example: {details['example']}, Complexity: {details['complexity']}"
        return "No suitable design pattern found."

# Example usage
def main():
    recommender = DesignPatternRecommender()

    purpose = input("Enter the purpose of the design pattern (e.g., 'create objects'): ")
    complexity = input("Enter the desired complexity level (e.g., 'low', 'medium', 'high'): ")

    recommendation = recommender.recommend_pattern(purpose=purpose, complexity=complexity)
    print(recommendation)

if __name__ == "__main__":
    main()
```