Q1. What is the primary goal of Object-Oriented Programming (OOP)?

The primary goal of **Object-Oriented Programming (OOP)** is to **organize and structure software in a way that makes it more modular, reusable, and easier to maintain**. This is achieved by modeling real-world entities as objects that encapsulate both **data** (attributes) and **behavior** (methods).

The key objectives of OOP include:

1. **Encapsulation**: Bundling the data (attributes) and methods (functions) that operate on the data into a single unit or class. This helps protect data from unauthorized access and reduces complexity.
2. **Abstraction**: Hiding the internal implementation details and exposing only the essential features of an object. This allows users to interact with objects without needing to understand the underlying complexity.
3. **Inheritance**: Allowing new classes to be based on existing ones, enabling code reuse and the creation of hierarchical relationships. Inheritance allows subclasses to inherit attributes and methods from their parent classes.
4. **Polymorphism**: Enabling objects of different classes to be treated as objects of a common superclass. This allows methods to be used interchangeably across different types of objects, enhancing flexibility.

In summary, the main goal of OOP is to improve software **modularity**, **reusability**, **flexibility**, and **maintainability**.

Q2. What is an object in Python?

In Python, an **object** is an instance of a class. It is a fundamental concept in Object-Oriented Programming (OOP), and it represents a specific entity that holds both **data** (attributes) and **behavior** (methods). Objects allow you to model real-world entities or concepts in code.

Key Characteristics of an Object in Python:

1. **State (Attributes)**: The data stored within the object. These are typically stored as variables within the object and are often referred to as *attributes* or *properties*. For example, in a `Car` class, an object might have attributes like `color`, `model`, and `speed`.
2. **Behavior (Methods)**: The actions or functions that an object can perform. These are defined within the class as *methods*. For example, a `Car` object might have methods like `accelerate()` or `brake()`.
3. **Identity**: Every object has a unique identity, which distinguishes it from other objects. In Python, this identity can be checked using the `id()` function, and each object exists independently in memory.

Example of an Object in Python:

```
# Defining a class
```

```
class Car:
    def __init__(self, make, model, color):
        self.make = make
        self.model = model
        self.color = color

    def start_engine(self):
        print(f"The {self.color} {self.make} {self.model}'s engine is now
running.")

# Creating an object (instance) of the Car class
my_car = Car("Toyota", "Corolla", "blue")

# Accessing attributes and calling methods on the object
print(my_car.make)  # Accessing the attribute
my_car.start_engine()  # Calling the method
```

In this example:

- `my_car` is an object of the `Car` class.
- The object `my_car` has attributes `make`, `model`, and `color`.
- The object has a behavior defined by the `start_engine()` method.

Summary:

An object in Python is a specific instance of a class, holding both **attributes** (data) and **methods** (functions), and it represents a real-world or conceptual entity within the program.

Q3. What is a class in Python?

In Python, a **class** is a blueprint or template for creating objects. It defines the structure (attributes) and behavior (methods) that the objects created from the class will have. A class serves as the foundation for creating instances (objects), encapsulating both data and functions that operate on that data.

Key Concepts of a Class in Python:

1. **Attributes**: These are variables that belong to the class or the instances of the class. Attributes store data that the objects created from the class will have.
2. **Methods**: These are functions defined within the class that describe the behavior of the objects created from the class. Methods can modify the attributes or perform actions related to the object.
3. **Constructor (`__init__`)**: This is a special method used to initialize new instances of the class. It is called automatically when a new object is created. The `__init__` method typically defines the initial state (values for attributes) of an object.
4. **Instance vs. Class Attributes**:
    - **Instance attributes** are specific to each object created from the class.
    - **Class attributes** are shared by all instances of the class and are defined directly within the class but outside any methods.

Example of a Class in Python:

```
# Defining a class
class Dog:
    # Constructor to initialize an instance of Dog
    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute

    # Method to simulate barking
    def bark(self):
        print(f"{self.name} is barking!")

    # Method to get the dog's age
    def get_age(self):
        return self.age

# Creating objects (instances) of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Bella", 5)

# Accessing attributes and calling methods on the objects
print(dog1.name)  # Output: Buddy
dog1.bark()       # Output: Buddy is barking!

print(dog2.get_age())  # Output: 5
```

Explanation:

- **Class Definition**: The `Dog` class is defined with a constructor `__init__`, which takes `name` and `age` as arguments. These are used to initialize the object's attributes (`self.name` and `self.age`).
- **Object Creation**: `dog1` and `dog2` are instances (objects) of the `Dog` class. Each object has its own values for `name` and `age`.
- **Methods**: `bark()` and `get_age()` are methods that define behaviors for the objects. The `bark()` method prints a message, and `get_age()` returns the dog's age.

Summary:

A **class** in Python is a template for creating objects, defining their structure (attributes) and behaviors (methods). Objects are instances of classes, and each object created from a class can have unique data while sharing common behavior defined in the class.

Q4. What are attributes and methods in a class?

In Python, **attributes** and **methods** are two fundamental components of a class, defining the state and behavior of the objects (instances) created from that class.

1. Attributes

Attributes are variables that belong to a class or an instance of a class. They store data or state information that describes an object. Attributes can be divided into two types:

- **Instance Attributes**: These are attributes specific to an instance (object) of the class. They are usually defined in the constructor method (`__init__`), and each object gets its own copy of these attributes.
- **Class Attributes**: These are attributes shared by all instances of the class. They are defined directly within the class and not inside any method.

Example:

```
class Dog:
    # Class attribute
    species = "Canine"  # All dogs share this value

    def __init__(self, name, age):
        # Instance attributes
        self.name = name
        self.age = age

# Creating objects
dog1 = Dog("Buddy", 3)
dog2 = Dog("Bella", 5)

print(dog1.species)  # Output: Canine (shared by all dogs)
print(dog2.species)  # Output: Canine (shared by all dogs)

print(dog1.name)  # Output: Buddy (specific to dog1)
print(dog2.name)  # Output: Bella (specific to dog2)
```

In this example:

- **`species`** is a **class attribute**, shared by all instances.
- **`name`** and **`age`** are **instance attributes**, specific to each object (e.g., `dog1` and `dog2`).

2. Methods

Methods are functions defined inside a class that describe the behaviors or actions that objects of the class can perform. Methods can modify an object's attributes, perform calculations, or interact with other objects.

- **Instance Methods**: These are methods that operate on instance data (attributes). The first parameter of an instance method is always `self`, which refers to the instance calling the method.
- **Class Methods**: These are methods that operate on class-level data (class attributes). They are defined with the `@classmethod` decorator and take `cls` as the first parameter.
- **Static Methods**: These are methods that don't operate on either instance or class attributes. They are defined with the `@staticmethod` decorator.

Example:

```
class Dog:
```

```
    species = "Canine"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age    # Instance attribute

    # Instance method
    def bark(self):
        print(f"{self.name} is barking!")

    # Instance method
    def get_age(self):
        return self.age

    # Class method
    @classmethod
    def get_species(cls):
        return cls.species

    # Static method
    @staticmethod
    def common_behavior():
        print("All dogs bark!")

# Creating objects
dog1 = Dog("Buddy", 3)

# Calling instance methods
dog1.bark()  # Output: Buddy is barking!
print(dog1.get_age())  # Output: 3

# Calling class method
print(Dog.get_species())  # Output: Canine

# Calling static method
Dog.common_behavior()  # Output: All dogs bark!
```

In this example:

- **`bark()`** and **`get_age()`** are **instance methods** that operate on the instance's attributes.
- **`get_species()`** is a **class method** that operates on class-level data.
- **`common_behavior()`** is a **static method**, which doesn't depend on the instance or class and is just a general behavior applicable to all instances.

Summary:

- **Attributes** represent the **state** of an object (data) and are stored as variables inside a class. They can be either **instance attributes** (unique to each object) or **class attributes** (shared across all objects).
- **Methods** represent the **behavior** of an object and are functions defined inside a class. They can modify or access an object's attributes, perform actions, or even interact with other objects.

Q5. What is the difference between class variables and instance variables in Python?

In Python, **class variables** and **instance variables** are both used to store data, but they differ in how they are defined and used. Here's a breakdown of their differences:

1. Instance Variables

- **Definition**: Instance variables are attributes that are tied to a specific instance (object) of a class. Each object has its own copy of these variables.
- **Scope**: Instance variables are defined inside the `__init__` method (or other methods) and are prefixed with `self` to indicate they belong to a specific instance.
- **Access**: They can be accessed through an object and are typically used to store data that can vary between different instances of the same class.

Example of Instance Variables:

```
class Dog:
    def __init__(self, name, age):
        self.name = name  # Instance variable
        self.age = age    # Instance variable

# Creating objects (instances)
dog1 = Dog("Buddy", 3)
dog2 = Dog("Bella", 5)

# Accessing instance variables
print(dog1.name)  # Output: Buddy
print(dog2.age)   # Output: 5
```

In this example:

- `self.name` and `self.age` are **instance variables**. Each dog object has its own `name` and `age`.

2. Class Variables

- **Definition**: Class variables are attributes that are shared by all instances of a class. They are defined within the class but outside any methods.
- **Scope**: Class variables belong to the class itself, not individual instances. They are accessed through the class name or through an object, but changes to a class variable affect all instances.
- **Access**: Class variables are typically used for values that are common to all instances of the class.

Example of Class Variables:

```
class Dog:
    species = "Canine"  # Class variable (shared by all instances)

    def __init__(self, name, age):
        self.name = name  # Instance variable
        self.age = age    # Instance variable

# Creating objects (instances)
dog1 = Dog("Buddy", 3)
dog2 = Dog("Bella", 5)

# Accessing class variable
```

```
print(Dog.species)   # Output: Canine (accessing via class)
print(dog1.species)  # Output: Canine (accessing via instance)
print(dog2.species)  # Output: Canine (accessing via instance)
```

In this example:

- `species` is a **class variable**. It is shared by all instances of the `Dog` class.

Key Differences:

Aspect	Instance Variables	Class Variables
Definition	Defined within the `__init__` method using `self`	Defined directly inside the class but outside any methods
Scope	Unique to each instance (object)	Shared by all instances of the class
Access	Accessed using `self` (e.g., `self.name`)	Accessed using the class name or through an instance (e.g., `Dog.species` or `dog1.species`)
Modification	Changing an instance variable affects only that specific instance	Changing a class variable affects all instances of the class
Usage	Used to store data that varies between objects	Used to store data that is common to all objects

Example to Illustrate the Difference:

```
class Dog:
    species = "Canine"  # Class variable (shared by all instances)

    def __init__(self, name, age):
        self.name = name  # Instance variable
        self.age = age    # Instance variable

# Creating objects (instances)
dog1 = Dog("Buddy", 3)
dog2 = Dog("Bella", 5)

# Changing the class variable
Dog.species = "Feline"

print(dog1.species)  # Output: Feline (shared class variable)
print(dog2.species)  # Output: Feline (shared class variable)

# Changing instance variables
dog1.name = "Max"
dog1.age = 4
```

```
print(dog1.name)  # Output: Max (instance variable specific to dog1)
print(dog2.name)  # Output: Bella (instance variable specific to dog2)
```

In this example:

- The **class variable** `species` is shared by all instances of the `Dog` class. Changing it through the class (`Dog.species = "Feline"`) affects all instances.
- The **instance variables** `name` and `age` are specific to each instance (e.g., `dog1` and `dog2`), so modifying `dog1.name` doesn't affect `dog2.name`.

Summary:

- **Instance variables** are tied to a specific instance and can have different values for different objects of the class.
- **Class variables** are shared by all instances of the class, and changes to a class variable affect all instances.

Q6. What is the purpose of the self parameter in Python class methods?

In Python, the **`self`** parameter in class methods is used to refer to the **instance** of the class that the method is being called on. It is the first parameter of instance methods and allows the method to access the instance's attributes and other methods. The `self` parameter is not a keyword, but a convention, and you can technically name it anything, though `self` is widely used by convention.

Purpose of the `self` Parameter:

1. **Access Instance Variables**: The `self` parameter allows instance methods to access and modify the instance-specific attributes (variables) of the object. It ensures that each instance has its own set of attributes and that methods can operate on these attributes.
2. **Access Other Methods in the Class**: Using `self`, an instance method can call other methods in the same class. It allows one method to interact with or modify the state of the object through the instance.
3. **Distinguish Between Instance and Class Variables**: The `self` parameter differentiates instance variables (specific to an object) from class variables (shared across all instances). By using `self`, the method can access and modify instance variables.

Example of the `self` Parameter:

```
class Dog:
    def __init__(self, name, age):
        self.name = name  # Instance variable
        self.age = age    # Instance variable

    def bark(self):
        print(f"{self.name} is barking!")
```

```
    def get_age(self):
        return self.age

    def birthday(self):
        self.age += 1  # Modifying an instance variable using self
        print(f"Happy Birthday {self.name}! Now you're {self.age} years
old.")

# Creating an instance (object) of the Dog class
dog1 = Dog("Buddy", 3)

# Calling instance methods
dog1.bark()  # Output: Buddy is barking!
print(dog1.get_age())  # Output: 3

# Calling a method that modifies the instance's age
dog1.birthday()  # Output: Happy Birthday Buddy! Now you're 4 years old.
```

Breakdown:

- **`self.name`** and **`self.age`** are instance variables. Each object (like `dog1`) has its own `name` and `age`.
- The `bark()`, `get_age()`, and `birthday()` methods all take `self` as the first parameter, allowing them to access and modify the attributes of the specific object they are called on.

Why is `self` Needed?

1. **Identification of the Current Instance**: The `self` parameter allows methods to know which specific object they are working with. Without it, methods wouldn't have a way to distinguish between different instances of a class. For example, `dog1.name` refers to the `name` attribute of the `dog1` object, not some global variable.
2. **Consistency Across All Instances**: If multiple objects are created from the same class, `self` ensures that the methods operate on the attributes that belong to the specific object that invoked the method.

Key Points:

- **`self` is not a keyword**: It is just a name for the first parameter of instance methods. You can technically use any name for it, but `self` is the conventional name.
- **`self` refers to the instance**: It ensures that methods can access and modify instance-specific data and interact with other methods within the class.
- **Not required for static or class methods**: `self` is only used in instance methods. For **class methods** and **static methods**, the first parameter is `cls` (for class methods) or no parameter at all (for static methods).

Summary:

The **`self`** parameter is crucial for defining instance methods in Python classes. It allows these methods to access and modify instance-specific data (attributes), ensuring that each object has its own state and behavior. Without `self`, methods wouldn't be able to refer to or differentiate between individual instances of a class.

Q7.

```
class Book:
    def __init__(self, title, author, isbn, publication_year, available_copies):
        self.title = title
        self.author = author
        self.isbn = isbn
        self.publication_year = publication_year
        self.available_copies = available_copies

    def check_out(self):
        if self.available_copies > 0:
            self.available_copies -= 1
            print(f"Book checked out. Available copies: {self.available_copies}")
        else:
            print("No available copies to check out.")

    def return_book(self):
        self.available_copies += 1
        print(f"Book returned. Available copies: {self.available_copies}")

    def display_book_info(self):
```

```
        print(f"Title: {self.title}")
        print(f"Author: {self.author}")
        print(f"ISBN: {self.isbn}")
        print(f"Publication Year: {self.publication_year}")
        print(f"Available Copies: {self.available_copies}")

# Example usage:
book = Book("The Great Gatsby", "F. Scott Fitzgerald", "9780743273565", 1925, 3)
book.display_book_info()
book.check_out()
book.return_book()
```

Q.8

```
class Ticket:
    def __init__(self, ticket_id, event_name, event_date, venue, seat_number, price):
        self.ticket_id = ticket_id
        self.event_name = event_name
        self.event_date = event_date
        self.venue = venue
        self.seat_number = seat_number
        self.price = price
        self.is_reserved = False

    def reserve_ticket(self):
        if not self.is_reserved:
```

```
            self.is_reserved = True
            print(f"Ticket reserved for {self.event_name} at {self.venue}. Seat: {self.seat_number}")
        else:
            print("Ticket is already reserved.")

    def cancel_reservation(self):
        if self.is_reserved:
            self.is_reserved = False
            print(f"Reservation for ticket {self.ticket_id} cancelled.")
        else:
            print("Ticket is not reserved.")

    def display_ticket_info(self):
        print(f"Ticket ID: {self.ticket_id}")
        print(f"Event Name: {self.event_name}")
        print(f"Event Date: {self.event_date}")
        print(f"Venue: {self.venue}")
        print(f"Seat Number: {self.seat_number}")
        print(f"Price: {self.price}")
        print(f"Reservation Status: {'Reserved' if self.is_reserved else 'Available'}")

# Example usage:
ticket = Ticket(1, "Concert", "2024-12-20", "Arena", "A12", 50)
```

```
ticket.display_ticket_info()

ticket.reserve_ticket()

ticket.cancel_reservation()
```

Q.9

```
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add_item(self, item):
        self.items.append(item)
        print(f"Item {item} added to the cart.")

    def remove_item(self, item):
        if item in self.items:
            self.items.remove(item)
            print(f"Item {item} removed from the cart.")
        else:
            print(f"Item {item} not found in the cart.")

    def view_cart(self):
        print("Items in your cart:")
        for item in self.items:
            print(item)
```

```
    def clear_cart(self):
        self.items = []
        print("All items removed from the cart.")

# Example usage:
cart = ShoppingCart()
cart.add_item("Laptop")
cart.add_item("Phone")
cart.view_cart()
cart.remove_item("Laptop")
cart.clear_cart()
```

Q10.

```
class Student:
    def __init__(self, name, age, grade, student_id):
        self.name = name
        self.age = age
        self.grade = grade
        self.student_id = student_id
        self.attendance = {}

    def update_attendance(self, date, status):
        self.attendance[date] = status
```

```
        print(f"Attendance for {self.name} on {date} updated to {status}.")

    def get_attendance(self):
        return self.attendance

    def get_average_attendance(self):
        total_days = len(self.attendance)
        if total_days == 0:
            return 0
        present_days = sum(1 for status in self.attendance.values() if status == 'present')
        return (present_days / total_days) * 100

# Example usage:
student = Student("John Doe", 16, "10th Grade", "S123")
student.update_attendance("2024-12-01", "present")
student.update_attendance("2024-12-02", "absent")
student.update_attendance("2024-12-03", "present")
print(student.get_attendance())
print(f"Average Attendance: {student.get_average_attendance()}%")
```