Constructor :

1. What is a constructor in Python? Explain its purpose and usage.
2. Differentiate between a parameterless constructor and a parameterized constructor in Python.
3. How do you define a constructor in a Python class? Provide an example.
4. Explain the `__init__` method in Python and its role in constructors.
5. In a class named `Person`, create a constructor that initializes the `name` and `age` attributes. Provide an example of creating an object of this class.
6. How can you call a constructor explicitly in Python? Give an example.
7. What is the significance of the `self` parameter in Python constructors? Explain with an example.
8. Discuss the concept of default constructors in Python. When are they used?
9. Create a Python class called `Rectangle` with a constructor that initializes the `width` and `height` attributes. Provide a method to calculate the area of the rectangle.
10. How can you have multiple constructors in a Python class? Explain with an example.
11. What is method overloading, and how is it related to constructors in Python?
12. Explain the use of the `super()` function in Python constructors. Provide an example.
13. Create a class called `Book` with a constructor that initializes the `title`, `author`, and `published_year` attributes. Provide a method to display book details.
14. Discuss the differences between constructors and regular methods in Python classes.
15. Explain the role of the `self` parameter in instance variable initialization within a constructor.
16. How do you prevent a class from having multiple instances by using constructors in Python? Provide an example.
17. Create a Python class called `Student` with a constructor that takes a list of subjects as a parameter and initializes the `subjects` attribute.
18. What is the purpose of the `__del__` method in Python classes, and how does it relate to constructors?
19. Explain the use of constructor chaining in Python. Provide a practical example.
20. Create a Python class called `Car` with a default constructor that initializes the `make` and `model` attributes. Provide a method to display car information.

Inheritance —

1. What is inheritance in Python? Explain its significance in object-oriented programming.
2. Differentiate between single inheritance and multiple inheritance in Python. Provide examples for each.
3. Create a Python class called `Vehicle` with attributes `color` and `speed`. Then, create a child class called `Car` that inherits from `Vehicle` and adds a `brand` attribute. Provide an example of creating a `Car` object.
4. Explain the concept of method overriding in inheritance. Provide a practical example.
5. How can you access the methods and attributes of a parent class from a child class in Python? Give an example.
6. Discuss the use of the `super()` function in Python inheritance. When and why is it used? Provide an example.

7. Create a Python class called `Animal` with a method `speak()`. Then, create child classes `Dog` and `Cat` that inherit from `Animal` and override the `speak()` method. Provide an example of using these classes.
8. Explain the role of the `isinstance()` function in Python and how it relates to inheritance.
9. What is the purpose of the `issubclass()` function in Python? Provide an example.
10. Discuss the concept of constructor inheritance in Python. How are constructors inherited in child classes?
11. Create a Python class called `Shape` with a method `area()` that calculates the area of a shape. Then, create child classes `Circle` and `Rectangle` that inherit from `Shape` and implement the `area()` method accordingly. Provide an example.
12. Explain the use of abstract base classes (ABCs) in Python and how they relate to inheritance. Provide an example using the `abc` module.
13. How can you prevent a child class from modifying certain attributes or methods inherited from a parent class in Python?
14. Create a Python class called `Employee` with attributes `name` and `salary`. Then, create a child class `Manager` that inherits from `Employee` and adds an attribute `department`. Provide an example.
15. Discuss the concept of method overloading in Python inheritance. How does it differ from method overriding?
16. Explain the purpose of the `__init__()` method in Python inheritance and how it is utilized in child classes.
17. Create a Python class called `Bird` with a method `fly()`. Then, create child classes `Eagle` and `Sparrow` that inherit from `Bird` and implement the `fly()` method differently. Provide an example of using these classes.
18. What is the "diamond problem" in multiple inheritance, and how does Python address it?
19. Discuss the concept of "is-a" and "has-a" relationships in inheritance, and provide examples of each.
20. Create a Python class hierarchy for a university system. Start with a base class `Person` and create child classes `Student` and `Professor`, each with their own attributes and methods. Provide an example of using these classes in a university context.

Encapsulation —

1. Explain the concept of encapsulation in Python. What is its role in object-oriented programming?
2. Describe the key principles of encapsulation, including access control and data hiding.
3. How can you achieve encapsulation in Python classes? Provide an example.
4. Discuss the difference between public, private, and protected access modifiers in Python.
5. Create a Python class called `Person` with a private attribute `__name`. Provide methods to get and set the name attribute.
6. Explain the purpose of getter and setter methods in encapsulation. Provide examples.

7. What is name mangling in Python, and how does it affect encapsulation?
8. Create a Python class called `BankAccount` with private attributes for the account balance (`__balance`) and account number (`__account_number`). Provide methods for depositing and withdrawing money.
9. Discuss the advantages of encapsulation in terms of code maintainability and security.
10. How can you access private attributes in Python? Provide an example demonstrating the use of name mangling.
11. Create a Python class hierarchy for a school system, including classes for students, teachers, and courses, and implement encapsulation principles to protect sensitive information.
12. Explain the concept of property decorators in Python and how they relate to encapsulation.
13. What is data hiding, and why is it important in encapsulation? Provide examples.
14. Create a Python class called `Employee` with private attributes for salary (`__salary`) and employee ID (`__employee_id`). Provide a method to calculate yearly bonuses.
15. Discuss the use of accessors and mutators in encapsulation. How do they help maintain control over attribute access?
16. What are the potential drawbacks or disadvantages of using encapsulation in Python?
17. Create a Python class for a library system that encapsulates book information, including titles, authors, and availability status.
18. Explain how encapsulation enhances code reusability and modularity in Python programs.
19. Describe the concept of information hiding in encapsulation. Why is it essential in software development?
20. Create a Python class called `Customer` with private attributes for customer details like name, address, and contact information. Implement encapsulation to ensure data integrity and security.

Polymorphism:-

1. What is polymorphism in Python? Explain how it is related to object-oriented programming.
2. Describe the difference between compile-time polymorphism and runtime polymorphism in Python.

3. Create a Python class hierarchy for shapes (e.g., circle, square, triangle) and demonstrate polymorphism through a common method, such as `calculate_area()`.
4. Explain the concept of method overriding in polymorphism. Provide an example.
5. How is polymorphism different from method overloading in Python? Provide examples for both.
6. Create a Python class called `Animal` with a method `speak()`. Then, create child classes like `Dog`, `Cat`, and `Bird`, each with their own `speak()` method. Demonstrate polymorphism by calling the `speak()` method on objects of different subclasses.
7. Discuss the use of abstract methods and classes in achieving polymorphism in Python. Provide an example using the `abc` module.
8. Create a Python class hierarchy for a vehicle system (e.g., car, bicycle, boat) and implement a polymorphic `start()` method that prints a message specific to each vehicle type.
9. Explain the significance of the `isinstance()` and `issubclass()` functions in Python polymorphism.
10. What is the role of the `@abstractmethod` decorator in achieving polymorphism in Python? Provide an example.
11. Create a Python class called `Shape` with a polymorphic method `area()` that calculates the area of different shapes (e.g., circle, rectangle, triangle).
12. Discuss the benefits of polymorphism in terms of code reusability and flexibility in Python programs.
13. Explain the use of the `super()` function in Python polymorphism. How does it help call methods of parent classes?
14. Create a Python class hierarchy for a banking system with various account types (e.g., savings, checking, credit card) and demonstrate polymorphism by implementing a common `withdraw()` method.
15. Describe the concept of operator overloading in Python and how it relates to polymorphism. Provide examples using operators like `+` and `*`.
16. What is dynamic polymorphism, and how is it achieved in Python?
17. Create a Python class hierarchy for employees in a company (e.g., manager, developer, designer) and implement polymorphism through a common `calculate_salary()` method.
18. Discuss the concept of function pointers and how they can be used to achieve polymorphism in Python.
19. Explain the role of interfaces and abstract classes in polymorphism, drawing comparisons between them.
20. Create a Python class for a zoo simulation, demonstrating polymorphism with different animal types (e.g., mammals, birds, reptiles) and their behavior (e.g., eating, sleeping, making sounds).

Abstraction:

1. What is abstraction in Python, and how does it relate to object-oriented programming?
2. Describe the benefits of abstraction in terms of code organization and complexity reduction.
3. Create a Python class called `Shape` with an abstract method `calculate_area()`. Then, create child classes (e.g., `Circle`, `Rectangle`) that implement the `calculate_area()` method. Provide an example of using these classes.
4. Explain the concept of abstract classes in Python and how they are defined using the `abc` module. Provide an example.
5. How do abstract classes differ from regular classes in Python? Discuss their use cases.
6. Create a Python class for a bank account and demonstrate abstraction by hiding the account balance and providing methods to deposit and withdraw funds.
7. Discuss the concept of interface classes in Python and their role in achieving abstraction.
8. Create a Python class hierarchy for animals and implement abstraction by defining common methods (e.g., `eat()`, `sleep()`) in an abstract base class.
9. Explain the significance of encapsulation in achieving abstraction. Provide examples.
10. What is the purpose of abstract methods, and how do they enforce abstraction in Python classes?
11. Create a Python class for a vehicle system and demonstrate abstraction by defining common methods (e.g., `start()`, `stop()`) in an abstract base class.
12. Describe the use of abstract properties in Python and how they can be employed in abstract classes.
13. Create a Python class hierarchy for employees in a company (e.g., manager, developer, designer) and implement abstraction by defining a common `get_salary()` method.
14. Discuss the differences between abstract classes and concrete classes in Python, including their instantiation.
15. Explain the concept of abstract data types (ADTs) and their role in achieving abstraction in Python.
16. Create a Python class for a computer system, demonstrating abstraction by defining common methods (e.g., `power_on()`, `shutdown()`) in an abstract base class.
17. Discuss the benefits of using abstraction in large-scale software development projects.
18. Explain how abstraction enhances code reusability and modularity in Python programs.
19. Create a Python class for a library system, implementing abstraction by defining common methods (e.g., `add_book()`, `borrow_book()`) in an abstract base class.
20. Describe the concept of method abstraction in Python and how it relates to polymorphism.

Composition 👍

1. Explain the concept of composition in Python and how it is used to build complex objects from simpler ones.
2. Describe the difference between composition and inheritance in object-oriented programming.
3. Create a Python class called `Author` with attributes for name and birthdate. Then, create a `Book` class that contains an instance of `Author` as a composition. Provide an example of creating a `Book` object.
4. Discuss the benefits of using composition over inheritance in Python, especially in terms of code flexibility and reusability.
5. How can you implement composition in Python classes? Provide examples of using composition to create complex objects.
6. Create a Python class hierarchy for a music player system, using composition to represent playlists and songs.
7. Explain the concept of "has-a" relationships in composition and how it helps design software systems.
8. Create a Python class for a computer system, using composition to represent components like CPU, RAM, and storage devices.
9. Describe the concept of "delegation" in composition and how it simplifies the design of complex systems.
10. Create a Python class for a car, using composition to represent components like the engine, wheels, and transmission.
11. How can you encapsulate and hide the details of composed objects in Python classes to maintain abstraction?
12. Create a Python class for a university course, using composition to represent students, instructors, and course materials.
13. Discuss the challenges and drawbacks of composition, such as increased complexity and potential for tight coupling between objects.
14. Create a Python class hierarchy for a restaurant system, using composition to represent menus, dishes, and ingredients.
15. Explain how composition enhances code maintainability and modularity in Python programs.
16. Create a Python class for a computer game character, using composition to represent attributes like weapons, armor, and inventory.
17. Describe the concept of "aggregation" in composition and how it differs from simple composition.
18. Create a Python class for a house, using composition to represent rooms, furniture, and appliances.

19. How can you achieve flexibility in composed objects by allowing them to be replaced or modified dynamically at runtime?
20. Create a Python class for a social media application, using composition to represent users, posts, and comments.