1. What is a constructor in Python? Explain its purpose and usage.

A constructor in Python is a special method that is automatically invoked when an object of a class is created. It is used to initialize the attributes of the class, allowing you to set the initial state of the object.

The constructor method in Python is defined using the __init__ method.

2. Differentiate between a parameterless constructor and a parameterized constructor in Python.

- **Parameterless Constructor:** A constructor that does not take any parameters other than self. It is used to initialize the attributes of the class with default values.
- class MyClass:
- def __init__(self):
- self.name = "John"
- **Parameterized Constructor:** A constructor that takes parameters (other than self) and uses them to initialize the attributes of the class.
- class MyClass:
- def __init__(self, name):
- self.name = name

3. How do you define a constructor in a Python class? Provide an example.

In Python, the constructor is defined using the __init__ method. Here's an example:

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

# Creating an object of the Person class
person1 = Person("Alice", 30)
```

4. Explain the __init__ method in Python and its role in constructors.

The __init__ method is the constructor method in Python. It is called automatically when an object of the class is created. It initializes the object's attributes using the parameters passed during object creation. It allows you to set up the object's initial state.

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

# Creating an object of the class
person1 = Person("Bob", 25)
```

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.

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

# Creating an object of the Person class
person1 = Person("Alice", 30)
print(f"Name: {person1.name}, Age: {person1.age}")
```

6. How can you call a constructor explicitly in Python? Give an example.

In Python, constructors are called automatically when an object is created, but they can also be explicitly called through the class name by using the __init__ method directly, although this is not typical.

```
class MyClass:
    def __init__(self, value):
        self.value = value

# Explicit constructor call
obj = MyClass.__init__(object, 10)
print(obj.value)
```

Note: This is not commonly done, as Python uses automatic constructor calls when creating objects.

7. What is the significance of the self parameter in Python constructors? Explain with an example.

The self parameter represents the instance of the object. It allows the constructor to access and modify the instance attributes of the class.

```
class Person:
    def __init__(self, name, age):
        self.name = name  # self allows access to the instance
        self.age = age

person1 = Person("John", 40)
print(person1.name)  # Output: John
```

8. Discuss the concept of default constructors in Python. When are they used?

A default constructor is a constructor that does not take any parameters other than self. It is automatically invoked when an object is created, and it initializes instance variables with default values. It is typically used when no parameters are passed during object creation.

```
class MyClass:
```

```
    def __init__(self):
        self.value = 0  # Default value

obj = MyClass()
print(obj.value)  # Output: 0
```

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.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Create a rectangle object
rect = Rectangle(5, 10)
print(f"Area of the rectangle: {rect.area()}")  # Output: 50
```

10. How can you have multiple constructors in a Python class? Explain with an example.

Python doesn't support method overloading directly, but you can achieve similar behavior by using default arguments or class methods.

```
class MyClass:
    def __init__(self, x=None, y=None):
        if x is not None and y is not None:
            self.x = x
            self.y = y
        else:
            self.x = 0
            self.y = 0

    @classmethod
    def from_tuple(cls, values):
        return cls(values[0], values[1])
```

```
obj1 = MyClass(5, 10)  # Regular constructor
obj2 = MyClass.from_tuple((3, 6))  # Alternative constructor

print(obj1.x, obj1.y)  # Output: 5 10
print(obj2.x, obj2.y)  # Output: 3 6
```

11. What is method overloading, and how is it related to constructors in Python?

Method overloading in Python refers to having multiple methods with the same name but different parameters. Python doesn't support overloading in the traditional sense like other languages, but you can simulate it using default arguments or variable-length arguments.

For constructors, you can achieve overloading-like behavior by using default arguments or class methods to handle different initialization scenarios.

12. Explain the use of the super() function in Python constructors. Provide an example.

The super() function in Python is used to call methods from a parent class. It can be used in constructors to initialize the attributes of a parent class.

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

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Calling parent class constructor
        self.breed = breed

dog = Dog("Rex", "Golden Retriever")
print(dog.name)   # Output: Rex
print(dog.breed)  # Output: Golden Retriever
```

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.

```
class Book:
    def __init__(self, title, author, published_year):
        self.title = title
        self.author = author
        self.published_year = published_year

    def display_details(self):
        print(f"Title: {self.title}")
        print(f"Author: {self.author}")
```

```
        print(f"Published Year: {self.published_year}")

book1 = Book("1984", "George Orwell", 1949)
book1.display_details()
```

14. Discuss the differences between constructors and regular methods in Python classes.

- **Constructors:** These are special methods (e.g., __init__) called automatically when an object is created. They are used to initialize the object's attributes.
- **Regular Methods:** These are functions defined inside a class that perform actions or computations related to the object. They are called explicitly by the object.

15. Explain the role of the self parameter in instance variable initialization within a constructor.

The self parameter is used to refer to the current instance of the class. It is used to assign values to instance variables, which are specific to each object created from the class.

```
class Person:
    def __init__(self, name, age):
        self.name = name  # 'self.name' refers to the instance variable
        self.age = age

person1 = Person("John", 25)
print(person1.name)  # Output: John
```

16. How do you prevent a class from having multiple instances by using constructors in Python? Provide an example.

To ensure that a class has only one instance, you can implement the Singleton design pattern by overriding the __new__ method or using a class variable to store the instance.

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance

obj1 = Singleton()
obj2 = Singleton()
print(obj1 is obj2)  # Output: True
```

17. Create a Python class called Student with a constructor that takes a list of subjects as a parameter and initializes the subjects attribute.

```
class Student:
    def __init__(self, subjects):
        self.subjects = subjects

student1 = Student(["Math", "Science", "English"])
print(student1.subjects)  # Output: ['Math', 'Science', 'English']
```

18. What is the purpose of the __del__ method in Python classes, and how does it relate to constructors?

The __del__ method is called when an object is destroyed (i.e., when it goes out of scope or is deleted). It is used for cleanup operations like closing files or releasing resources. It is the opposite of the constructor (__init__).

```
class MyClass:
    def __init__(self):
        print("Object created")

    def __del__(self):
        print("Object destroyed")

obj = MyClass()
del obj  # Calls __del__
```

19. Explain the use of constructor chaining in Python. Provide a practical example.

Constructor chaining refers to calling another constructor from a subclass or within the same class. It is often used to reuse code from a parent constructor or another constructor within the same class.

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

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

dog = Dog("Rex", "Bulldog")
```

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.

```
class Car:
    def __init__(self, make="Toyota", model="Camry"):
        self.make = make
        self.model = model

    def display_info(self):
        print(f"Make: {self.make}")
        print(f"Model: {self.model}")

car1 = Car()
car1.display_info()  # Output: Make: Toyota, Model: Camry

car2 = Car("Honda", "Civic")
car2.display_info()  # Output: Make: Honda, Model: Civic
```

1. What is inheritance in Python? Explain its significance in object-oriented programming.

Inheritance in Python is a mechanism where a new class (child class) inherits attributes and methods from an existing class (parent class). It allows the child class to inherit functionality from the parent class and extend or modify it as needed. This promotes code reuse, improves maintainability, and establishes a relationship between classes.

Significance:

- Promotes code reuse.
- Allows modification of inherited methods (method overriding).
- Facilitates the creation of hierarchies, modeling real-world relationships.

2. Differentiate between single inheritance and multiple inheritance in Python. Provide examples for each.

- **Single Inheritance:** A class inherits from one parent class.
- class Animal:
- def speak(self):
- print("Animal speaks")
-
- class Dog(Animal): # Single inheritance
- def bark(self):

- print("Dog barks")
-
- dog = Dog()
- dog.speak() # Inherited method
- dog.bark()   # Dog's own method
- **Multiple Inheritance:** A class inherits from more than one parent class.
- class Animal:
- def speak(self):
- print("Animal speaks")
-
- class Flyable:
- def fly(self):
- print("Flying")
-
- class Bird(Animal, Flyable): # Multiple inheritance
- pass
-
- bird = Bird()
- bird.speak() # Inherited from Animal
- bird.fly()   # Inherited from Flyable

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.

```
class Vehicle:
    def __init__(self, color, speed):
        self.color = color
        self.speed = speed

class Car(Vehicle):
    def __init__(self, color, speed, brand):
        super().__init__(color, speed)  # Inherit from Vehicle
        self.brand = brand

car = Car("Red", 120, "Toyota")
```

```
print(f"Car Brand: {car.brand}, Color: {car.color}, Speed: {car.speed}")
```

4. Explain the concept of method overriding in inheritance. Provide a practical example.

Method Overriding occurs when a child class defines a method with the same name as one in the parent class, but provides a different implementation. It allows the child class to change the behavior of a method inherited from the parent class.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):  # Overriding the method
        print("Dog barks")

dog = Dog()
dog.speak()  # Output: Dog barks
```

5. How can you access the methods and attributes of a parent class from a child class in Python? Give an example.

You can use the super() function to access methods and attributes of the parent class from a child class.

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

    def speak(self):
        print(f"{self.name} makes a sound")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Access parent class constructor
        self.breed = breed

    def speak(self):
        super().speak()  # Call the parent class method
        print(f"{self.name} barks!")

dog = Dog("Rex", "Bulldog")
```

```
dog.speak()  # Output: Rex makes a sound\nRex barks!
```

6. Discuss the use of the super() function in Python inheritance. When and why is it used? Provide an example.

The super() function is used to call methods or access attributes from the parent class in the child class. It is helpful when you want to extend or modify the behavior of the parent class methods.

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

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Call parent constructor
        self.breed = breed

dog = Dog("Buddy", "Golden Retriever")
print(f"{dog.name} is a {dog.breed}")
```

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.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
        print("Cat meows")

dog = Dog()
dog.speak()  # Output: Dog barks

cat = Cat()
```

```
cat.speak()  # Output: Cat meows
```
8. Explain the role of the isinstance() function in Python and how it relates to inheritance.

The isinstance() function checks if an object is an instance of a class or a subclass thereof. It is useful for verifying the type of an object, especially when dealing with inheritance.

```
class Animal:
    pass

class Dog(Animal):
    pass

dog = Dog()
print(isinstance(dog, Animal))  # Output: True (dog is an instance of Animal)
```
9. What is the purpose of the issubclass() function in Python? Provide an example.

The issubclass() function checks if a class is a subclass of another class. It returns True if the first class is a subclass of the second class.

```
class Animal:
    pass

class Dog(Animal):
    pass

print(issubclass(Dog, Animal))  # Output: True (Dog is a subclass of Animal)
```
10. Discuss the concept of constructor inheritance in Python. How are constructors inherited in child classes?

In Python, constructors are inherited by default, but the child class can define its own constructor (which can also call the parent's constructor using super()). If the child class doesn't define a constructor, the parent class's constructor is used.

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

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Inherit constructor
        self.breed = breed
```

```
dog = Dog("Rex", "Bulldog")
print(f"{dog.name} is a {dog.breed}")
```

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.

```
import math

class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

circle = Circle(5)
print(f"Circle Area: {circle.area()}")

rectangle = Rectangle(4, 6)
print(f"Rectangle Area: {rectangle.area()}")
```

12. Explain the use of abstract base classes (ABCs) in Python and how they relate to inheritance. Provide an example using the abc module.

Abstract Base Classes (ABCs) define methods that must be implemented by subclasses. ABCs cannot be instantiated directly and are used to ensure that subclasses implement certain methods.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

# shape = Shape()  # This would raise an error, as Shape is abstract
circle = Circle(5)
print(f"Circle Area: {circle.area()}")
```

13. How can you prevent a child class from modifying certain attributes or methods inherited from a parent class in Python?

To prevent modification, you can use the @property decorator for attributes to make them read-only or simply not define the method in the child class.

```
class Parent:
    def __init__(self):
        self._value = 10

    @property
    def value(self):
        return self._value

class Child(Parent):
    pass

child = Child()
print(child.value)  # Output: 10
```

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.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Manager(Employee):
    def __init__(self, name, salary, department):
        super().__init__(name, salary)
        self.department = department

manager = Manager("Alice", 75000, "HR")
print(f"Manager: {manager.name}, Salary: {manager.salary}, Department: {manager.department}")
```

15. Discuss the concept of method overloading in Python inheritance. How does it differ from method overriding?

Python does not support method overloading in the traditional sense, but you can achieve similar functionality using default arguments or variable-length arguments.

- **Method Overloading:** Defining methods with the same name but different parameters (not supported directly).
- **Method Overriding:** Redefining a method in the child class to provide a new implementation.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self, sound="Bark"):  # Overloading-like behavior using default argument
        print(f"Dog {sound}")

dog = Dog()
dog.speak()  # Output: Dog Bark
```

16. Explain the purpose of the __init__() method in Python inheritance and how it is utilized in child classes.

The __init__() method initializes attributes in an object. In inheritance, the child class can call the parent's __init__() method using super() to initialize inherited attributes.

```
class Animal:
```

```
    def __init__(self, name):
        self.name = name

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Calling parent constructor
        self.breed = breed

dog = Dog("Buddy", "Golden Retriever")
print(f"{dog.name} is a {dog.breed}")
```

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.

```
class Bird:
    def fly(self):
        pass

class Eagle(Bird):
    def fly(self):
        print("Eagle soars high in the sky")

class Sparrow(Bird):
    def fly(self):
        print("Sparrow flutters and hops")

eagle = Eagle()
eagle.fly()  # Output: Eagle soars high in the sky

sparrow = Sparrow()
sparrow.fly()  # Output: Sparrow flutters and hops
```

18. What is the "diamond problem" in multiple inheritance, and how does Python address it?

The "diamond problem" occurs when a class inherits from two classes that have a common ancestor, potentially causing ambiguity. Python resolves this using the **C3 Linearization** algorithm, ensuring a consistent order of method resolution.

```
class A:
```

```
    def speak(self):
        print("A speaks")

class B(A):
    def speak(self):
        print("B speaks")

class C(A):
    def speak(self):
        print("C speaks")

class D(B, C):
    pass

d = D()
d.speak()  # Output: B speaks (method resolution order)
```

19. Discuss the concept of "is-a" and "has-a" relationships in inheritance, and provide examples of each.

In object-oriented programming, the concepts of "is-a" and "has-a" relationships describe how classes are related to one another in terms of inheritance and composition.

"is-a" Relationship:

An "is-a" relationship is established when a class is a specialized version of another class, meaning the child class is a type of the parent class. This relationship is most commonly seen in inheritance, where the child class is a subclass of the parent class.

- **Example: If we have a class Animal, and we create subclasses Dog and Cat, then a Dog is a type of Animal, and a Cat is a type of Animal. The child classes inherit properties and behaviors from the parent class.**

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Woof!")
```

```
class Cat(Animal):
    def speak(self):
        print("Meow!")

# Usage
dog = Dog()
cat = Cat()
dog.speak()  # Output: Woof!
cat.speak()  # Output: Meow!
```

Here, both Dog and Cat "is-a" Animal, meaning they are specialized versions of the Animal class.

"has-a" Relationship:

A "has-a" relationship occurs when a class contains an object of another class as one of its attributes. This is a form of composition, where one class contains another class rather than inheriting from it.

- Example: A Car has-a Engine. The Car class does not extend Engine but contains an Engine object as an attribute.

```
class Engine:
    def start(self):
        print("Engine starts")

class Car:
    def __init__(self, engine):
        self.engine = engine  # "has-a" relationship

    def drive(self):
        self.engine.start()
        print("Car is driving")

# Usage
engine = Engine()
car = Car(engine)
car.drive()  # Output: Engine starts
             #         Car is driving
```

In this case, the Car "has-a" Engine, meaning the Car contains an Engine object, but Car is not a type of Engine.

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.

In this example, we will create a Person class, and then create subclasses Student and Professor to represent specific types of people in a university system. Each subclass will have its own attributes and methods.

Base Class: Person

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

    def get_details(self):
        return f"Name: {self.name}, Age: {self.age}"
```

Child Class: Student

```
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)  # Call the constructor of the parent class
        self.student_id = student_id

    def get_student_info(self):
        return f"Student ID: {self.student_id}, " + self.get_details()

    def study(self):
        print(f"{self.name} is studying.")
```

Child Class: Professor

```
class Professor(Person):
    def __init__(self, name, age, employee_id):
        super().__init__(name, age)
        self.employee_id = employee_id

    def get_professor_info(self):
        return f"Employee ID: {self.employee_id}, " + self.get_details()

    def teach(self):
```

```
        print(f"{self.name} is teaching.")
```

Example Usage:

```
# Create instances of Student and Professor
student = Student("Alice", 20, "S12345")
professor = Professor("Dr. Smith", 45, "P98765")

# Display information about student and professor
print(student.get_student_info())  # Output: Student ID: S12345, Name: Alice, Age: 20
print(professor.get_professor_info())  # Output: Employee ID: P98765, Name: Dr. Smith, Age: 45

# Call methods specific to Student and Professor
student.study()  # Output: Alice is studying.
professor.teach()  # Output: Dr. Smith is teaching.
```

Explanation of the University System Example:

1. **Person Class:**
    - This is the base class with common attributes like name and age. It also has a method get_details() that provides basic information about the person.
2. **Student Class:**
    - The Student class inherits from Person. It adds an additional attribute student_id and defines a method study() that is specific to students.
    - The get_student_info() method provides information about the student, including their ID and personal details, combining data from the Person class.
3. **Professor Class:**
    - The Professor class also inherits from Person and adds an attribute employee_id. It has a method teach() specific to professors.
    - The get_professor_info() method provides the professor's details, including their employee ID, using information from both the Professor and Person classes.

This hierarchy allows for the creation of specific types of people in the university (students and professors) while reusing common attributes and methods from the Person class.

1. Explain the concept of encapsulation in Python. What is its role in object-oriented programming?

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It refers to bundling the data (attributes) and methods (functions) that operate on the data into a single unit or class, and restricting access to certain components of the class. The purpose of encapsulation is to protect an object's internal state from unauthorized access and modification, ensuring better control over how data is accessed or modified.

In OOP, encapsulation helps:

- Protect internal object data.
- Enhance code maintainability.
- Simplify debugging and testing by providing controlled access to data.

2. Describe the key principles of encapsulation, including access control and data hiding.

- **Access Control:** Encapsulation allows defining access levels to the attributes and methods of a class. This can be controlled using access modifiers:
  - **Public**: Accessible by any other class.
  - **Protected**: Accessible within the class and its subclasses.
  - **Private**: Accessible only within the class itself.
- **Data Hiding:** The idea is to hide the internal state of the object and only expose the necessary functionalities. This prevents direct access to sensitive or critical data, ensuring it can only be modified through controlled methods (getters and setters).

3. How can you achieve encapsulation in Python classes? Provide an example.

In Python, encapsulation is achieved by defining attributes and methods inside a class and controlling access to them through naming conventions (e.g., using underscores). For private attributes, a double underscore (__) is used.

```
class Person:
    def __init__(self, name, age):
        self.__name = name  # Private attribute
        self.__age = age    # Private attribute

    # Getter method for name
    def get_name(self):
        return self.__name

    # Setter method for name
    def set_name(self, name):
        self.__name = name

    # Getter method for age
```

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

    # Setter method for age
    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Age cannot be negative!")

# Usage
person = Person("John", 30)
print(person.get_name())  # Output: John
person.set_name("Alice")
print(person.get_name())  # Output: Alice
```

4. Discuss the difference between public, private, and protected access modifiers in Python.

- **Public:** Attributes or methods are accessible from anywhere. In Python, there are no strict access modifiers, but attributes that are not prefixed with an underscore (_) are considered public by convention.
    - Example: self.name
- **Protected:** Intended to be accessed only within the class and subclasses. Prefixing with a single underscore (_) suggests that an attribute is protected, but it can still be accessed outside the class.
    - Example: self._name
- **Private:** Intended to be used only within the class. A double underscore (__) prefix makes the attribute private, making it harder (though not impossible) to access from outside the class due to name mangling.
    - Example: self.__name

5. Create a Python class called Person with a private attribute __name. Provide methods to get and set the name attribute.

```
class Person:
    def __init__(self, name):
        self.__name = name  # Private attribute

    def get_name(self):
        return self.__name  # Getter method
```

```
    def set_name(self, name):
        self.__name = name  # Setter method

# Usage
person = Person("John")
print(person.get_name())  # Output: John
person.set_name("Alice")
print(person.get_name())  # Output: Alice
```

6. Explain the purpose of getter and setter methods in encapsulation. Provide examples.

Getter methods provide controlled access to private attributes, allowing you to retrieve the value of a private attribute.

Setter methods allow you to set or modify the value of a private attribute while controlling how the data is modified (e.g., validation).

```
class Person:
    def __init__(self, name):
        self.__name = name

    # Getter method
    def get_name(self):
        return self.__name

    # Setter method
    def set_name(self, name):
        if name.isalpha():  # Validation check
            self.__name = name
        else:
            print("Invalid name!")

person = Person("John")
print(person.get_name())  # Output: John
person.set_name("Alice")  # Valid name
print(person.get_name())  # Output: Alice
person.set_name("123")    # Invalid name
```

7. What is name mangling in Python, and how does it affect encapsulation?

Name mangling is a technique used by Python to make private variables more difficult to access from outside the class. When a variable is prefixed with double underscores (__), Python internally changes the name of the variable to include the class name, making it harder to accidentally or intentionally access the attribute from outside the class.
For example, an attribute __name in a class Person would be internally changed to _Person__name. This doesn't prevent access, but it makes it more difficult.

```
class Person:
    def __init__(self, name):
        self.__name = name

person = Person("John")
print(person._Person__name)  # Accessing the private attribute using name mangling
```

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.

```
class BankAccount:
    def __init__(self, account_number, balance=0):
        self.__account_number = account_number  # Private attribute
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        else:
            print("Deposit amount must be positive!")

    def withdraw(self, amount):
        if amount > 0 and self.__balance >= amount:
            self.__balance -= amount
        else:
            print("Invalid withdrawal amount!")

    def get_balance(self):
        return self.__balance
```

```
# Usage
account = BankAccount("12345", 1000)
account.deposit(500)
account.withdraw(200)
print(account.get_balance())  # Output: 1300
```

9. Discuss the advantages of encapsulation in terms of code maintainability and security.

- **Code Maintainability:** Encapsulation allows you to modify the internal implementation of a class without affecting the external code that relies on it, as long as the public interface remains the same. This makes it easier to maintain and refactor code.
- **Security:** By hiding the internal state of an object and exposing only necessary functionality, encapsulation helps protect sensitive data from unintended modification or access, ensuring the integrity of the object.

10. How can you access private attributes in Python? Provide an example demonstrating the use of name mangling.

Private attributes can be accessed using name mangling, where the name of the attribute is changed to include the class name. This is not recommended but demonstrates how Python handles private variables internally.

```
class Person:
    def __init__(self, name):
        self.__name = name

person = Person("John")
print(person._Person__name)  # Accessing private attribute using 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.

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name

class Student(Person):
```

```
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.__student_id = student_id

    def get_student_id(self):
        return self.__student_id

class Teacher(Person):
    def __init__(self, name, age, employee_id):
        super().__init__(name, age)
        self.__employee_id = employee_id

    def get_employee_id(self):
        return self.__employee_id

class Course:
    def __init__(self, course_name, teacher):
        self.__course_name = course_name
        self.__teacher = teacher

    def get_course_name(self):
        return self.__course_name

# Example usage
student = Student("Alice", 20, "S12345")
teacher = Teacher("Mr. Smith", 45, "T98765")
course = Course("Mathematics", teacher)

print(student.get_name())  # Alice
print(student.get_student_id())  # S12345
print(course.get_course_name())  # Mathematics
```

12. Explain the concept of property decorators in Python and how they relate to encapsulation.

The property decorator in Python allows you to define methods that act like attributes. It provides controlled access to private attributes by letting you define getter, setter, and deleter methods for an attribute without having to explicitly call these methods.

```
class Person:
    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
        return self.__name  # Getter

    @name.setter
    def name(self, name):
        self.__name = name  # Setter

# Usage
person = Person("John")
print(person.name)  # Output: John
person.name = "Alice"
print(person.name)  # Output: Alice
```

13. What is data hiding, and why is it important in encapsulation? Provide examples.

Data hiding refers to restricting access to the internal state of an object, ensuring that its data can only be accessed or modified through well-defined methods. It prevents unauthorized or unintended changes to the object's state, ensuring the integrity and security of the data.

```
class Account:
    def __init__(self, balance):
        self.__balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance
```

```
account = Account(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

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.

```
class Employee:
    def __init__(self, name, salary, employee_id):
        self.__name = name
        self.__salary = salary
        self.__employee_id = employee_id

    def calculate_bonus(self):
        return self.__salary * 0.1  # 10% bonus

# Usage
employee = Employee("Alice", 50000, "E1234")
print(f"Bonus: {employee.calculate_bonus()}")  # Output: 5000
```

15. Discuss the use of accessors and mutators in encapsulation. How do they help maintain control over attribute access?

Accessors (Getters) allow you to read the value of an attribute, while **mutators (Setters)** enable you to modify the attribute. These methods provide controlled access to the attributes, ensuring data integrity by allowing you to enforce rules or validation checks when attributes are accessed or modified.

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    # Accessor (Getter)
    def get_name(self):
        return self.__name

    # Mutator (Setter)
    def set_name(self, name):
        self.__name = name
```

```
person = Person("John", 30)
print(person.get_name())  # Output: John
person.set_name("Alice")
print(person.get_name())  # Output: Alice
```

16. What are the potential drawbacks or disadvantages of using encapsulation in Python?

While encapsulation provides significant advantages, there are some potential drawbacks:

- **Complexity:** It can add complexity to the code, especially when using multiple levels of access control and getter/setter methods.
- **Performance Overhead:** Using methods for attribute access can introduce slight performance overhead, though usually negligible.
- **Less flexibility:** In some cases, encapsulation can make it harder to directly manipulate attributes without going through getters/setters.

17. Create a Python class for a library system that encapsulates book information, including titles, authors, and availability status.

```
class Book:
    def __init__(self, title, author, available=True):
        self.__title = title
        self.__author = author
        self.__available = available

    def get_details(self):
        return f"Title: {self.__title}, Author: {self.__author}, Available: {self.__available}"

    def check_out(self):
        if self.__available:
            self.__available = False
        else:
            print("Book is not available.")

    def return_book(self):
        self.__available = True

# Usage
book = Book("The Great Gatsby", "F. Scott Fitzgerald")
print(book.get_details())
book.check_out()
print(book.get_details())
```

18. Explain how encapsulation enhances code reusability and modularity in Python programs.

Encapsulation allows classes to have well-defined interfaces for interacting with their data, which encourages modularity. By hiding the implementation details, encapsulated classes can be reused in different parts of the program without modifying their internal behavior. This reduces dependencies and enhances the flexibility of the code.

19. Describe the concept of information hiding in encapsulation. Why is it essential in software development?

Information hiding is a core principle of encapsulation, where the internal workings of a class (e.g., its data) are hidden from the outside world. By exposing only the necessary methods for interaction, you prevent unauthorized access and modification of critical data. This improves software maintainability, security, and reduces the risk of unintended errors.

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.

```
class Customer:
    def __init__(self, name, address, contact):
        self.__name = name
        self.__address = address
        self.__contact = contact

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name

    def get_address(self):
        return self.__address

    def set_address(self, address):
        self.__address = address

    def get_contact(self):
        return self.__contact

    def set_contact(self, contact):
        self.__contact = contact
```

```
# Usage
customer = Customer("Alice", "123 Main St", "555-1234")
print(customer.get_name())  # Output: Alice
customer.set_address("456 Elm St")
print(customer.get_address())  # Output: 456 Elm St
```

1. What is polymorphism in Python? Explain how it is related to object-oriented programming.

Polymorphism is a core concept of object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It means "many forms," allowing the same method or function to behave differently based on the object it is acting upon. This allows for flexibility and reusability in code, where the same method or function can operate on different types of objects.

In Python, polymorphism is typically achieved through method overriding (inherited methods are redefined in a subclass) or method overloading (though Python doesn't support method overloading in the traditional sense, you can simulate it using default arguments).

Example:

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def speak(self):
        print("Meow!")

# Demonstrating polymorphism
animals = [Dog(), Cat()]
for animal in animals:
    animal.speak()  # Output: Woof! Meow!
```

In this example, the speak method is called on objects of Dog and Cat, but the output varies, demonstrating polymorphism.

2. Describe the difference between compile-time polymorphism and runtime polymorphism in Python.

- **Compile-time polymorphism (Static polymorphism)**: This refers to the ability to overload methods or operators during compile time. Python doesn't support traditional compile-time polymorphism like some other languages (such as Java or C++), since Python does not have strict method signatures.
- **Runtime polymorphism (Dynamic polymorphism)**: This is the most common type of polymorphism in Python, where the method that gets called is determined at runtime based on the object. This is typically achieved through method overriding, where a subclass defines a method that overrides a method in its superclass.

Example of runtime polymorphism:

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def speak(self):
        print("Meow!")

# Demonstrating runtime polymorphism
def make_speak(animal: Animal):
    animal.speak()

make_speak(Dog())  # Output: Woof!
make_speak(Cat())  # Output: Meow!
```

Here, the method speak() is dynamically resolved based on the object passed to the make_speak() function, which demonstrates runtime polymorphism.

3. Create a Python class hierarchy for shapes (e.g., circle, square, triangle) and demonstrate polymorphism through a common method, such as calculate_area().

```
import math

class Shape:
    def calculate_area(self):
        raise NotImplementedError("Subclass must implement abstract method")
```

```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return math.pi * self.radius ** 2

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def calculate_area(self):
        return self.side ** 2

class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def calculate_area(self):
        return 0.5 * self.base * self.height

# Demonstrating polymorphism
shapes = [Circle(5), Square(4), Triangle(3, 6)]
for shape in shapes:
    print(f"Area: {shape.calculate_area()}")  # Different calculations based on object type
```

Output:

Area: 78.53981633974483
Area: 16
Area: 9.0

In this example, all the shapes (circle, square, triangle) share the calculate_area() method, but each one implements the method in its own way, demonstrating polymorphism.

4. Explain the concept of method overriding in polymorphism. Provide an example.

Method overriding occurs when a subclass provides its own implementation of a method that is already defined in its superclass. This allows the subclass to modify or extend the behavior of the inherited method.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):  # Method overriding
        print("Woof!")

class Cat(Animal):
    def speak(self):  # Method overriding
        print("Meow!")

# Demonstrating method overriding
dog = Dog()
cat = Cat()
dog.speak()  # Output: Woof!
cat.speak()  # Output: Meow!
```

In this example, both Dog and Cat override the speak() method of the Animal class.

5. How is polymorphism different from method overloading in Python? Provide examples for both.

- **Polymorphism**: Polymorphism allows methods to behave differently based on the object calling them. It is achieved through method overriding, where a method in a subclass has the same name as the method in the parent class but provides a different implementation.
- **Method overloading**: Method overloading refers to defining multiple methods with the same name but different parameters. In Python, traditional method overloading (like in Java or C++) is not supported directly. However, Python can simulate overloading by using default arguments or variable-length arguments.

Example of method overloading simulation using default arguments:

```
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c
```

```
calc = Calculator()
print(calc.add(2, 3))  # Output: 5
print(calc.add(2, 3, 4))  # Output: 9
```

Example of polymorphism:

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def speak(self):
        print("Meow!")

# Demonstrating polymorphism
animals = [Dog(), Cat()]
for animal in animals:
    animal.speak()  # Output: Woof! Meow!
```

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.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def speak(self):
        print("Meow!")
```

```
class Bird(Animal):
    def speak(self):
        print("Chirp!")

# Demonstrating polymorphism
animals = [Dog(), Cat(), Bird()]
for animal in animals:
    animal.speak()  # Output: Woof! Meow! Chirp!
```

7. Discuss the use of abstract methods and classes in achieving polymorphism in Python. Provide an example using the abc module.

Abstract classes and methods allow us to define methods that must be implemented by subclasses, enforcing a certain interface. This ensures that subclasses can implement the method in their own way, achieving polymorphism.

Example:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius ** 2

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def calculate_area(self):
        return self.side ** 2
```

```
# Demonstrating polymorphism
shapes = [Circle(5), Square(4)]
for shape in shapes:
    print(f"Area: {shape.calculate_area()}")  # Different calculations based on object type
```