oops

September 30, 2024

OOPS

[1]:
```python
# Constructor
#1
class Person:
    def __init__(self, name, age):
        # Initialize instance variables
        self.name = name
        self.age = age

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

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

# Calling the method to display information
person1.display_info()  # Output: Name: Alice, Age: 30
```

```
Name: Alice, Age: 30
```

[1]:
```python
#2 Differentiate between a parameterless constructor and a parameterized␣
 ↪constructor in Python.
Parameterless Constructor: A constructor without any parameters other than self.
 ↪ It initializes the object with default values.

python
Copy code
class Example:
    def __init__(self):
        self.value = 0
Parameterized Constructor: A constructor that takes one or more parameters␣
 ↪(other than self) to initialize the object with specific values.

python
Copy code
class Example:
    def __init__(self, value):
```

```
        self.value = value
```

John

[3]:
```
#3 How do you define a constructor in a Python class? Provide an example.
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} says Woof!"


my_dog = Dog("Buddy", 3)


print(my_dog.name)
print(my_dog.age)
print(my_dog.bark())
```

Buddy
3
Buddy says Woof!

[5]:
```
# 4. Explain the `__init__` method in Python and its role in constructors.
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
    def display_info(self):
        return f"{self.year} {self.make} {self.model}"

my_car = Car("Toyota", "Corolla", 2020)

print(my_car.display_info())  # Output: 2020 Toyota Corolla
```

2020 Toyota Corolla

[7]:
```
# 5 . In a class named `Person`, create a constructor that initializes the␣
↪`name` and `age` attributes. Provide anexample of creating an object of this␣
↪class.
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def introduce(self):
```

```python
        return f"Hello, my name is {self.name} and I am {self.age} years old."

person1 = Person("Alice", 30)


print(person1.name)
print(person1.age)
print(person1.introduce())
```

```
Alice
30
Hello, my name is Alice and I am 30 years old.
```

[9]:
```python
# 6.How can you call a constructor explicitly in Python? Give an examples.
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def display_info(self):
        return f"{self.make} {self.model}"

car1 = Car("Toyota", "Camry")
car2 = Car.__init__(car1, "Honda", "Civic")


print(car1.display_info())
print(car1.display_info())
```

```
Honda Civic
Honda Civic
```

[11]:
```python
# 7. What is the significance of the `self` parameter in Python constructors?␣
↪Explain with an example.
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def greet(self):
        return f"Hi, I'm {self.name} and I'm {self.age} years old."

person1 = Person("John", 25)

print(person1.greet())
```

```
Hi, I'm John and I'm 25 years old.
```

[13]:
```python
# 8
class Dog:
    def __init__(self, name="Unknown", age=0):
        self.name = name
        self.age = age

dog1 = Dog("Buddy", 5)
dog2 = Dog()

print(dog1.name, dog1.age)
print(dog2.name, dog2.age)
```

```
Buddy 5
Unknown 0
```

[15]:
```python
# 9
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

rect = Rectangle(5, 10)

print("Area of the rectangle:", rect.area())  # Output:
```

```
Area of the rectangle: 50
```

[17]:
```python
# 10
class Circle:
    def __init__(self, radius=1):
        self.radius = radius

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

circle1 = Circle(3)
circle2 = Circle()

print("Area of circle1:", circle1.area())
print("Area of circle2:", circle2.area())
```

```
Area of circle1: 28.274333882308138
Area of circle2: 3.141592653589793
```

[19]:
```python
# 11
class Person:
    def __init__(self, name, age=None):
        self.name = name
        self.age = age if age is not None else "Not specified"

# Creating instances
person1 = Person("Alice", 30)
person2 = Person("Bob")

print(person1.age)
print(person2.age)
```

```
30
Not specified
```

[21]:
```python
# 12
class Animal:
    def __init__(self, species):
        self.species = species

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

dog = Dog("Buddy", 5)

print(dog.species)
print(dog.name)
print(dog.age)
```

```
Dog
Buddy
5
```

[23]:
```python
# 13
class Book:
    def __init__(self, title, author, published_year):
        self.title = title
        self.author = author
        self.published_year = published_year

    def display_details(self):
        return f"Title: {self.title}\nAuthor: {self.author}\nPublished Year:␣
 ↪{self.published_year}"
```

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

print(book1.display_details())
```

```
Title: 1984
Author: George Orwell
Published Year: 1949
```

[ ]:
```
# 14. Discuss the differences between constructors and regular methods in␣
↪Python classes.
Constructors and regular methods serve different purposes in Python classes:

Purpose:

Constructor (__init__): Used to initialize an instance of a class. It is called␣
↪automatically when a new object is created.
Regular Methods: Perform specific operations or actions related to the class␣
↪but are called explicitly after an object is created.
Naming:

Constructor: Always named __init__.
Regular Methods: Can have any valid method name, defined by the programmer.
Parameters:

Constructor: Always takes self as the first parameter and can take additional␣
↪parameters for initialization.
Regular Methods: Also take self as the first parameter, but they can have␣
↪various parameters depending on the method's functionality.
Return Value:

Constructor: Does not return a value. It implicitly returns None.
Regular Methods: Can return values as needed.
```

[25]:
```
# 15. Role of the self Parameter in Instance Variable Initialization
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
car1 = Car("Toyota", "Corolla")
print(car1.make)
```

```
Toyota
```

[27]:
```
# 16.  Preventing Multiple Instances of a Class
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
```

```python
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance

    def __init__(self, value):
        self.value = value

singleton1 = Singleton(1)
singleton2 = Singleton(2)

print(singleton1.value)
print(singleton2.value)
print(singleton1 is singleton2)
```

```
2
2
True
```

[29]:

```python
# 17.  Student Class with Subjects
class Student:
    def __init__(self, subjects):
        self.subjects = subjects

    def display_subjects(self):
        return f"Subjects: {', '.join(self.subjects)}"

student1 = Student(["Math", "Science", "History"])

print(student1.display_subjects())
```

```
Subjects: Math, Science, History
```

[31]:

```python
# 18. Purpose of the __del__ Method in Python Classes
class Resource:
    def __init__(self):
        print("Resource acquired.")

    def __del__(self):
        print("Resource released.")


res = Resource()
del res
```

```
Resource acquired.
Resource released.
```

[33]:
```python
# 19. Constructor Chaining in Python
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

class Car(Vehicle):
    def __init__(self, make, model, year):
        super().__init__(make, model)
        self.year = year

    def display_info(self):
        return f"{self.year} {self.make} {self.model}"

car = Car("Toyota", "Corolla", 2020)

print(car.display_info())
```

```
2020 Toyota Corolla
```

[35]:
```python
# 20. Car Class with Default Constructor
class Car:
    def __init__(self, make="Unknown", model="Unknown"):
        self.make = make
        self.model = model

    def display_info(self):
        return f"Car Make: {self.make}, Model: {self.model}"

car1 = Car("Honda", "Civic")
car2 = Car()

print(car1.display_info())
print(car2.display_info())
```

```
Car Make: Honda, Model: Civic
Car Make: Unknown, Model: Unknown
```

[ ]:
```python
# Inheritance:
# 1. What is inheritance in Python? Explain its significance in object-oriented␣
 ↪prog
Inheritance is a fundamental concept in object-oriented programming (OOP) that␣
 ↪allows a class (known as a child or subclass) to inherit attributes and␣
 ↪methods from another class (known as a parent or superclass). This promotes␣
 ↪code reusability and establishes a hierarchical relationship between classes.

Significance of Inheritance:
```

```
Code Reusability: Allows you to use existing code in new classes, reducing␣
↪redundancy.
Hierarchy: Helps in organizing classes into a logical structure, making the␣
↪code easier to manage.
Polymorphism: Enables the use of the same method in different classes,␣
↪enhancing flexibility.
Extensibility: Facilitates extending the functionality of existing classes␣
↪without modifying them.
```

[39]:
```
# 2.  Differentiate between single inheritance and multiple inheritance in␣
↪Python. Provide examples for each.
#Single Inheritance
#In single inheritance, a class (child class) inherits from one superclass only.
↪ This creates a straightforward relationship.
class Animal:
    def speak(self):
        return "Animal speaks"

class Dog(Animal):
    def bark(self):
        return "Woof!"


dog = Dog()
print(dog.speak())
print(dog.bark())
#Multiple Inheritance
#In multiple inheritance, a class can inherit from more than one superclass.␣
↪This allows the child class to access attributes and methods from multiple␣
↪parent classes.
class Flyer:
    def fly(self):
        return "Flying"

class Swimmer:
    def swim(self):
        return "Swimming"

class Duck(Flyer, Swimmer):
    def quack(self):
        return "Quack!"

duck = Duck()
print(duck.fly())
print(duck.swim())
print(duck.quack())
```

```
Animal speaks
Woof!
Flying
Swimming
Quack!
```

[41]:
```python
# 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

    def display_info(self):
        return f"Color: {self.color}, Speed: {self.speed} km/h"

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

    def display_car_info(self):
        return f"{self.brand} Car - {self.display_info()}"

my_car = Car("Red", 150, "Toyota")

print(my_car.display_car_info())
```

```
Toyota Car - Color: Red, Speed: 150 km/h
```

[43]:
```python
# 4 Explain the concept of method overriding in inheritance. Provide a
↪practical example
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def speak(self):  # Overriding the speak method
        return "Woof!"

class Cat(Animal):
    def speak(self):  # Overriding the speak method
        return "Meow!"

# Using the classes
dog = Dog()
```

```
cat = Cat()

print(dog.speak())  # Output: Woof!
print(cat.speak())  # Output: Meow!
```

```
Woof!
Meow!
```

[45]:
```
# 5. How can you access the methods and attributes of a parent class from a␣
↪child class in Python? Give anexample.
class Vehicle:
    def __init__(self, color):
        self.color = color

    def display_color(self):
        return f"Color: {self.color}"

class Car(Vehicle):
    def __init__(self, color, brand):
        super().__init__(color)  # Call the parent constructor
        self.brand = brand

    def display_info(self):
        return f"{self.brand} - {self.display_color()}"

# Creating an instance of Car
my_car = Car("Blue", "Ford")

# Accessing methods and attributes
print(my_car.display_info())  # Output: Ford - Color: Blue
```

```
Ford - Color: Blue
```

[47]:
```
# 6 .  Discuss the use of the `super()` function in Python inheritance. When␣
↪and why is it used? Provide anexample
class Person:
    def __init__(self, name):
        self.name = name

class Student(Person):
    def __init__(self, name, student_id):
        super().__init__(name)  # Calling the parent constructor
        self.student_id = student_id

    def display_info(self):
        return f"Name: {self.name}, Student ID: {self.student_id}"

# Creating an instance of Student
```

```
student = Student("Alice", "S12345")

# Displaying student information
print(student.display_info())  # Output: Name: Alice, Student ID: S12345
```

```
Name: Alice, Student ID: S12345
```

[49]:
```
# 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):
        return "Some sound"

class Dog(Animal):
    def speak(self):  # Overriding the speak method
        return "Woof!"

class Cat(Animal):
    def speak(self):  # Overriding the speak method
        return "Meow!"

# Using the classes
dog = Dog()
cat = Cat()

print(dog.speak())  # Output: Woof!
print(cat.speak())  # Output: Meow!
```

```
Woof!
Meow!
```

[51]:
```
# 8. Explain the role of the `isinstance()` function in Python and how it
↪relates to inheritance.

class Animal:
    pass

class Dog(Animal):
    pass

dog = Dog()

print(isinstance(dog, Dog))      # Output: True
print(isinstance(dog, Animal))   # Output: True
print(isinstance(dog, str))      # Output: False
```

```
True
```

```
True
False
```

[53]:
```python
# 9.What is the purpose of the `issubclass()` function in Python? Provide an␣
↪example.
class Animal:
    pass

class Dog(Animal):
    pass

class Cat(Animal):
    pass

print(issubclass(Dog, Animal))
print(issubclass(Cat, Animal))
print(issubclass(Dog, Cat))
```

```
True
True
False
```

[57]:
```python
# 10. . Discuss the concept of constructor inheritance in Python. How are␣
↪constructors inherited in child classes?
class Vehicle:
    def __init__(self, color):
        self.color = color

class Car(Vehicle):
    def __init__(self, color, brand):
        super().__init__(color)  # Calling the parent constructor
        self.brand = brand

# Creating an instance of Car
my_car = Car("Red", "Toyota")
print(my_car.color)  # Output: Red
print(my_car.brand)  # Output: Toyota
```

```
Red
Toyota
```

[55]:
```python
# 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()`␣
↪methodaccordingly. Provide an example.
import math

class Shape:
```

```
    def area(self):
        raise NotImplementedError("Subclasses must implement this method.")

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)
rectangle = Rectangle(4, 6)

print(f"Area of Circle: {circle.area():.2f}")
print(f"Area of Rectangle: {rectangle.area()}")
```

```
Area of Circle: 78.54
Area of Rectangle: 24
```

[59]:

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

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)

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

```python
        self.height = height

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

circle = Circle(5)
rectangle = Rectangle(4, 6)

print(f"Area of Circle: {circle.area():.2f}")
print(f"Area of Rectangle: {rectangle.area()}")
```

```
Area of Circle: 78.54
Area of Rectangle: 24
```

[61]:
```python
# 13. How can you prevent a child class from modifying certain attributes or␣
↪methods inherited from a parentclass in Python?
class Base:
    def __init__(self):
        self.__private_var = 42
        self._protected_var = 24

    @property
    def private_var(self):
        return self.__private_var

class Derived(Base):
    def modify(self):
        # self.__private_var = 100
        self._protected_var = 30

base = Base()
print(base.private_var)
```

```
42
```

[63]:
```python
# 14
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)  # Call the parent constructor
        self.department = department

manager = Manager("Alice", 80000, "Sales")
```

```
print(f"Name: {manager.name}, Salary: {manager.salary}, Department: {manager.
↪department}")
```

```
Name: Alice, Salary: 80000, Department: Sales
```

[65]:
```
# 15.
class Example:
    def show(self, a=None):
        if a is not None:
            print(f"Value: {a}")
        else:
            print("No value provided")

obj = Example()
obj.show()
obj.show(10)
```

```
No value provided
Value: 10
```

[ ]:
```
# 16.
class Animal:
    def __init__(self, species):
        self.species = species

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

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

[67]:
```
# 17.
class Bird:
    def fly(self):
        raise NotImplementedError("Subclasses must implement this method.")

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

class Sparrow(Bird):
    def fly(self):
        return "Sparrow flutters quickly!"

eagle = Eagle()
```

```
sparrow = Sparrow()

print(eagle.fly())
print(sparrow.fly())
```

```
Eagle soars high in the sky!
Sparrow flutters quickly!
```

[69]:
```
# 18.
class A:
    def greet(self):
        return "Hello from A"

class B(A):
    def greet(self):
        return "Hello from B"

class C(A):
    def greet(self):
        return "Hello from C"

class D(B, C):
    pass

d = D()
print(d.greet())
print(D.mro())
```

```
Hello from B
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
'__main__.A'>, <class 'object'>]
```

[4]:
```
# 19. Discuss the concept of "is-a" and "has-a" relationships in inheritance,
↪and provide examples of each.
#Is-a" Relationship
#The "is-a" relationship describes inheritance in object-oriented programming.
↪It indicates that a subclass is a specialized version of a superclass.
↪Essentially, the subclass inherits properties and behaviors from the
↪superclass.
class Vehicle:
    def start_engine(self):
        return "Engine started"

class Car(Vehicle):
    def play_music(self):
        return "Playing music"

class Bike(Vehicle):
```

```
    def ring_bell(self):
        return "Ring ring!"

# "Has-a" Relationship
# The "has-a" relationship indicates composition, where one class contains␣
↪instances of another class as part of its attributes. This relationship␣
↪emphasizes that one object is composed of one or more objects.
class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower

class Wheel:
    def __init__(self, size):
        self.size = size

class Car:
    def __init__(self, engine, wheels):
        self.engine = engine
        self.wheels = wheels

    def describe(self):
        return f"This car has an engine with {self.engine.horsepower}␣
↪horsepower and {len(self.wheels)} wheels."
```

[2]:
```
# 20.  Create a Python class hierarchy for a university system. Start with a␣
↪base class `Person` and create childclasses `Student` and `Professor`, each␣
↪with their own attributes and methods. Provide an example of usingthese␣
↪classes in a university context
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        return f"Hi, I'm {self.name}, and I'm {self.age} years old."


class Student(Person):
    def __init__(self, name, age, student_id, major):
        super().__init__(name, age)
        self.student_id = student_id
        self.major = major

    def study(self):
        return f"{self.name} is studying {self.major}."

    def introduce(self):
```

```python
        return f"Hi, I'm {self.name}, a {self.major} major."


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

    def teach(self):
        return f"Professor {self.name} is teaching in the {self.department}␣
 ↪department."

    def introduce(self):
        return f"Hi, I'm Professor {self.name}, and I teach in the {self.
 ↪department} department."


if __name__ == "__main__":
    student = Student("Alice", 20, "S123", "Computer Science")
    professor = Professor("Dr. Smith", 45, "E456", "Mathematics")

    print(student.introduce())
    print(student.study())

    print(professor.introduce())
    print(professor.teach())
```

```
Hi, I'm Alice, a Computer Science major.
Alice is studying Computer Science.
Hi, I'm Professor Dr. Smith, and I teach in the Mathematics department.
Professor Dr. Smith is teaching in the Mathematics department.
```

[ ]:
```python
#  Encapsulation:
#1. Explain the concept of encapsulation in Python. What is its role in␣
 ↪object-oriented programming?
Encapsulation is a fundamental concept in object-oriented programming (OOP)␣
 ↪that restricts direct access to an object's internal state and behavior. In␣
 ↪Python, encapsulation helps in bundling the data (attributes) and methods␣
 ↪(functions) that operate on that data into a single unit, or class. This␣
 ↪concept helps maintain control over the data, reducing the risk of␣
 ↪unintended interference and misuse.
```

[ ]:
```python
# 2. Describe the key principles of encapsulation, including access control and␣
 ↪data hiding.
# Access Control: Encapsulation restricts access to certain components of an␣
 ↪object. This is done to protect the integrity of the object's data.
```

```
# Data Hiding: By hiding the internal state of an object, encapsulation ensures␣
↪that the object can only be modified in well-defined ways, typically through␣
↪public methods.
```

[6]:
```
# 3. How can you achieve encapsulation in Python classes? Provide an example
class BankAccount:
    def __init__(self, account_number):
        self.__account_number = account_number
        self.__balance = 0.0

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

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return amount
        return None

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

[ ]:
```
#. Access Modifiers in Python
Public: Attributes or methods that can be accessed from outside the class (e.g.
↪, self.balance).

Protected: Attributes or methods that are intended to be protected from outside␣
↪access but can be accessed in subclasses (e.g., _balance).

Private: Attributes or methods that cannot be accessed directly from outside␣
↪the class. They are prefixed with double underscores (e.g., __balance).
```

[4]:
```
#5.Create a Python class called `Person` with a private attribute `__name`.␣
↪Provide methods to get and set thename attribute
class Person:
    def __init__(self, name):
        self.__name = name

    def get_name(self):
        """Getter method to access the private attribute __name."""
        return self.__name

    def set_name(self, name):
        """Setter method to modify the private attribute __name."""
        if isinstance(name, str) and len(name) > 0:
```

```python
            self.__name = name
        else:
            raise ValueError("Name must be a non-empty string.")

if __name__ == "__main__":
    person = Person("Alice")
    print(person.get_name())

    person.set_name("Bob")
    print(person.get_name())
```

```
Alice
Bob
```

[1]:

```python
# 6.
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, new_name):
        if isinstance(new_name, str) and new_name:
            self._name = new_name
        else:
            raise ValueError("Name must be a non-empty string")

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, new_age):
        if isinstance(new_age, int) and new_age >= 0:
            self._age = new_age
        else:
            raise ValueError("Age must be a non-negative integer")

person = Person("Alice", 30)
print(person.name)
print(person.age)
```

```python
person.name = "Bob"
person.age = 31

try:
    person.age = -1
except ValueError as e:
    print(e)
```

```
Alice
30
Age must be a non-negative integer
```

[3]:
```python
# 7.  What is name mangling in Python, and how does it affect encapsula
class Base:
    def __init__(self):
        self.__value = "Base Value"

class Derived(Base):
    def __init__(self):
        super().__init__()
        self.__value = "Derived Value"

base = Base()
derived = Derived()
print(base._Base__value)
print(derived._Derived__value)
```

```
Base Value
Derived Value
```

[5]:
```python
# 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: str, initial_balance: float = 0.0):
        self.__account_number = account_number
        self.__balance = initial_balance

    def deposit(self, amount: float):
        if amount <= 0:
            raise ValueError("Deposit amount must be positive.")
        self.__balance += amount
        print(f"Deposited: ${amount:.2f}. New balance: ${self.__balance:.2f}")

    def withdraw(self, amount: float):
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive.")
```

```python
        if amount > self.__balance:
            raise ValueError("Insufficient funds.")
        self.__balance -= amount
        print(f"Withdrew: ${amount:.2f}. New balance: ${self.__balance:.2f}")

    def get_balance(self) -> float:
        return self.__balance

    def get_account_number(self) -> str:
        return self.__account_number

if __name__ == "__main__":
    account = BankAccount("123456789", 1000.0)

    print(f"Account Number: {account.get_account_number()}")
    print(f"Initial Balance: ${account.get_balance():.2f}")

    account.deposit(500.0)
    account.withdraw(200.0)

    try:
        account.withdraw(1500.0)
    except ValueError as e:
        print(e)
```

```
Account Number: 123456789
Initial Balance: $1000.00
Deposited: $500.00. New balance: $1500.00
Withdrew: $200.00. New balance: $1300.00
Insufficient funds.
```

[ ]:
```python
# 9. Discuss the advantages of encapsulation in terms of code maintainability
↪and security.
Advantages of Encapsulation
1. Code Maintainability
Modular Design: By encapsulating data and behavior within classes, code is
↪organized into distinct modules. This modularity makes it easier to
↪understand and manage individual components without needing to comprehend
↪the entire system.

Reduced Complexity: Encapsulation allows developers to hide complex
↪implementation details and expose only necessary interfaces. This
↪abstraction simplifies the interaction with the code, making it easier to
↪maintain and update.
```

```
Ease of Refactoring: Changes to the internal implementation of a class can be
  made with minimal impact on other parts of the codebase. As long as the
  public interface remains consistent, modifications to the internal logic
  won't break existing functionality.

Improved Debugging: Encapsulated code can be easier to debug since issues can
  be isolated within specific classes. Developers can focus on individual
  components without needing to trace through unrelated code.

2. Security
Controlled Access: Encapsulation restricts direct access to an object's
  internal state. By using private and protected attributes, developers can
  control how data is accessed and modified, reducing the risk of unintended
  changes.

Data Validation: Through methods (getters and setters), you can implement
  validation logic to ensure that only valid data is assigned to attributes.
  This guards against erroneous or malicious input, enhancing data integrity.

Increased Robustness: By limiting the ways in which data can be manipulated,
  encapsulation helps prevent bugs and vulnerabilities. This makes the
  codebase more robust against unintended usage patterns.

Abstraction of Sensitive Information: Sensitive data can be protected by
  encapsulating it in a way that it's not directly accessible. This is
  particularly useful in scenarios like banking applications, where access to
  account information must be tightly controlled.
```

[7]:
```python
# 10.. How can you access private attributes in Python? Provide an example
  demonstrating the use of name mangling.
class Sample:
    def __init__(self):
        self.__private_attr = "I am private"

    def get_private_attr(self):
        return self.__private_attr

sample_instance = Sample()

try:
    print(sample_instance.__private_attr)
except AttributeError as e:
    print(e)

print(sample_instance._Sample__private_attr)

print(sample_instance.get_private_attr())
```

```
'Sample' object has no attribute '__private_attr'
I am private
I am private
```

[9]:
```python
# 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: str, age: int):
        self.__name = name
        self.__age = age

    def get_name(self) -> str:
        return self.__name

    def get_age(self) -> int:
        return self.__age


class Student(Person):
    def __init__(self, name: str, age: int, student_id: str):
        super().__init__(name, age)
        self.__student_id = student_id
        self.__grades = []

    def add_grade(self, grade: float):
        if 0 <= grade <= 100:
            self.__grades.append(grade)
        else:
            raise ValueError("Grade must be between 0 and 100.")

    def get_average_grade(self) -> float:
        if not self.__grades:
            return 0.0
        return sum(self.__grades) / len(self.__grades)

    def get_student_id(self) -> str:
        return self.__student_id


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

    def get_employee_id(self) -> str:
        return self.__employee_id
```

```python
    def teach_course(self, course: 'Course'):
        print(f"{self.get_name()} is teaching {course.get_title()}.")


class Course:
    def __init__(self, title: str, code: str):
        self.__title = title
        self.__code = code

    def get_title(self) -> str:
        return self.__title

    def get_code(self) -> str:
        return self.__code


if __name__ == "__main__":
    student = Student("Alice", 20, "S12345")
    student.add_grade(85.0)
    student.add_grade(90.0)

    teacher = Teacher("Mr. Smith", 35, "T98765")
    course = Course("Mathematics", "MATH101")

    print(f"Student Name: {student.get_name()}")
    print(f"Student ID: {student.get_student_id()}")
    print(f"Average Grade: {student.get_average_grade():.2f}")

    print(f"Teacher Name: {teacher.get_name()}")
    print(f"Employee ID: {teacher.get_employee_id()}")
    teacher.teach_course(course)
```

```
Student Name: Alice
Student ID: S12345
Average Grade: 87.50
Teacher Name: Mr. Smith
Employee ID: T98765
Mr. Smith is teaching Mathematics.
```

[11]:
```python
# 12.  Explain the concept of property decorators in Python and how they relate␣
↪to encapsulation.
class Circle:
    def __init__(self, radius: float):
        self.__radius = radius

    @property
```

```
    def radius(self) -> float:
        """Getter for radius."""
        return self.__radius

    @radius.setter
    def radius(self, value: float):
        """Setter for radius with validation."""
        if value < 0:
            raise ValueError("Radius cannot be negative.")
        self.__radius = value

    @property
    def area(self) -> float:
        """Calculates area based on radius."""
        return 3.14159 * (self.__radius ** 2)

# Example usage
if __name__ == "__main__":
    circle = Circle(5)
    print(f"Radius: {circle.radius}")
    print(f"Area: {circle.area}")

    circle.radius = 10  # Changing the radius
    print(f"New Radius: {circle.radius}")
    print(f"New Area: {circle.area}")

    try:
        circle.radius = -5
    except ValueError as e:
        print(e)
```

```
Radius: 5
Area: 78.53975
New Radius: 10
New Area: 314.159
Radius cannot be negative.
```

[13]:

```
# 13. What is data hiding, and why is it important in encapsulation? Provide␣
↪examples.
class BankAccount:
    def __init__(self, account_number: str, initial_balance: float = 0.0):
        self.__account_number = account_number
        self.__balance = initial_balance

    def deposit(self, amount: float):
        if amount <= 0:
            raise ValueError("Deposit amount must be positive.")
```

```python
        self.__balance += amount

    def withdraw(self, amount: float):
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive.")
        if amount > self.__balance:
            raise ValueError("Insufficient funds.")
        self.__balance -= amount

    def get_balance(self) -> float:
        return self.__balance


if __name__ == "__main__":
    account = BankAccount("123456789", 1000.0)

    print(f"Initial Balance: ${account.get_balance():.2f}")

    account.deposit(500.0)
    print(f"Balance after deposit: ${account.get_balance():.2f}")

    try:
        account.withdraw(2000.0)
    except ValueError as e:
        print(e)

    try:
        print(account.__balance)
    except AttributeError as e:
        print(e)
```

```
Initial Balance: $1000.00
Balance after deposit: $1500.00
Insufficient funds.
'BankAccount' object has no attribute '__balance'
```

[1]:
```python
# 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, salary, employee_id):
        self.__salary = salary
        self.__employee_id = employee_id

    def calculate_bonus(self, bonus_percentage):
        bonus = self.__salary * (bonus_percentage / 100)
        return bonus
```

```
    def get_salary(self):
        return self.__salary

    def get_employee_id(self):
        return self.__employee_id
```

[ ]:

```
# 15.  Discuss the use of accessors and mutators in encapsulation. How do they help maintain control overattribute access?
Role of Accessors and Mutators in Encapsulation:
Accessors (Getters):

Definition: Accessor methods allow external code to read the value of private attributes in a controlled manner.
Purpose: Instead of exposing the attribute directly, the accessor method provides a way to retrieve the value, ensuring control over how it is accessed. It can also add logic to validate access or format the data when it's returned.
Example:
python
Copy code
def get_salary(self):
    return self.__salary
Mutators (Setters):

Definition: Mutator methods allow external code to modify the value of private attributes while ensuring that the modification follows certain rules or conditions.
Purpose: By using mutators, you can control how attributes are modified. For example, you can enforce validations, restrict certain types of changes, or trigger side effects (like updating dependent attributes).
Example:
python
Copy code
def set_salary(self, new_salary):
    if new_salary > 0:  # Basic validation to ensure positive salary
        self.__salary = new_salary
    else:
        raise ValueError("Salary must be positive.")
Benefits of Using Accessors and Mutators:
Controlled Access:

You can restrict who gets to see or modify certain attributes. By controlling access through methods, you ensure that object data is not directly exposed or manipulated arbitrarily.
Validation and Error Handling:
```

```
Mutators allow you to add logic to check the validity of data before it's changed. For instance, ensuring that a salary is always positive or that a bonus percentage falls within a certain range.
Data Integrity:

Accessors and mutators help maintain the integrity of your object's state by preventing incorrect or unsafe modifications. They can ensure that the internal data remains consistent, even after changes.
Decoupling of Implementation and Interface:

By using methods to access or modify data, you can change the internal representation of attributes without affecting external code. For example, you could store a salary in one currency but return it in another through an accessor, without changing the external interface.
Read-Only and Write-Only Attributes:

By defining only getters or setters, you can create attributes that are read-only (can be accessed but not changed) or write-only (can be changed but not accessed directly).
Example:
Here's an example showing how accessors and mutators help with validation and control:

python
Copy code
class Employee:
    def __init__(self, salary, employee_id):
        self.__salary = salary
        self.__employee_id = employee_id

    # Accessor (getter) for salary
    def get_salary(self):
        return self.__salary

    # Mutator (setter) for salary with validation
    def set_salary(self, new_salary):
        if new_salary > 0:
            self.__salary = new_salary
        else:
            raise ValueError("Salary must be positive.")

    # Accessor for employee ID
    def get_employee_id(self):
        return self.__employee_id

# Usage
employee = Employee(50000, "E123")
```

```
print(employee.get_salary())  # Accessing salary
employee.set_salary(55000)  # Modifying salary
print(employee.get_salary())  # Accessing updated salary
Conclusion:
Accessors and mutators are important tools in encapsulation that allow
 ↪developers to maintain strict control over how an object's attributes are
 ↪accessed or modified. By using these methods, you can enforce rules, ensure
 ↪data integrity, and provide a consistent interface, all while keeping the
 ↪internal structure hidden and protected from unintended manipulation.
```

[5]:
```
# What are the potential drawbacks or disadvantages of using encapsulation in
 ↪Python?
#Lack of True Private Variables:
#Explanation: In Python, "private" attributes (those prefixed with __) are not
 ↪truly private. Python uses name mangling to change the attribute's name
 ↪internally (e.g., __salary becomes _ClassName__salary), but this is not a
 ↪strict form of data hiding.
#Consequence: Developers can still access and modify these "private" attributes
 ↪by circumventing name mangling, which somewhat weakens the concept of
 ↪encapsulation.
employee = Employee(50000, "E123")
print(employee._Employee__salary)  # Accessing the 'private' attribute
#Overhead and Boilerplate Code:
# Explanation: Encapsulation often requires creating getter and setter methods
 ↪for each attribute. This can result in additional boilerplate code that adds
 ↪complexity without much immediate benefit, especially for simple or small
 ↪classes.
# Consequence: It increases the size of the codebase, potentially making it
 ↪harder to maintain and read, particularly when attributes do not require
 ↪additional validation.
def get_salary(self):
    return self.__salary

def set_salary(self, new_salary):
    if new_salary > 0:
        self.__salary = new_salary
    else:
        raise ValueError("Salary must be positive.")
#Performance Overhead:
#Explanation: Using accessors and mutators introduces method calls every time
 ↪an attribute is accessed or modified. This may create a slight performance
 ↪overhead compared to direct access, especially in performance-critical
 ↪applications where the getter/setter is called frequently.
#Consequence: For highly performant code, the overhead introduced by
 ↪encapsulation might be unnecessary, though Python generally handles this
 ↪efficiently for most applications.
```

```python
#4. Reduced Flexibility for Simple Classes:
#Explanation: For simple data structures (like plain data objects or value␣
↪types), strict encapsulation can be overkill. Python’s philosophy encourages␣
↪simplicity and readability, so using encapsulation for classes that just␣
↪hold data (without additional behavior) may complicate the code␣
↪unnecessarily.
#Consequence: In cases where attributes are straightforward and don’t require␣
↪validation, directly exposing them (via public attributes) might be a more␣
↪Pythonic and efficient approach.
#Example: Using a plain attribute in Python instead of defining a getter and␣
↪setter.

class Employee:
    def __init__(self, salary, employee_id):
        self.salary = salary
        self.employee_id = employee_id  # No need for encapsulation here
# 5. Misuse of Getters and Setters:
# Explanation: In some cases, encapsulation is overused or misused. Developers␣
↪may create getter and setter methods even when there is no real need for␣
↪them (i.e., when the attributes do not need validation or transformation).
# Consequence: This results in unnecessarily complex code. Python does not␣
↪require the use of getters/setters for attributes by default, and overly␣
↪using them without justification can conflict with the language's simplicity␣
↪principles.
# 6. Increased Complexity for Inheritance:
# Explanation: When dealing with encapsulation, especially with private␣
↪attributes (prefixed with __), subclasses cannot directly access private␣
↪attributes of the parent class due to name mangling. This forces subclasses␣
↪to use accessors or other methods to modify or interact with these␣
↪attributes.
# Consequence: This can make inheritance more cumbersome, requiring workarounds␣
↪or breaking encapsulation to allow subclasses access to parent class␣
↪attributes.
# 7. Potential for Breaking Backward Compatibility:
# Explanation: If attributes are initially public and are later encapsulated (e.
↪g., made private and accessed via getter/setter methods), it can break␣
↪backward compatibility with any existing code that accessed those attributes␣
↪directly.
# Consequence: Existing code that depends on direct attribute access may fail␣
↪if attributes are encapsulated later, requiring refactoring or adding␣
↪compatibility layers.
# 8. Pythonic Alternatives (Properties):
# Explanation: Python offers a more elegant alternative through properties,␣
↪which allows attributes to be accessed like regular variables while still␣
↪providing encapsulation. However, this can make traditional getter/setter␣
↪methods feel less Pythonic and unnecessary.
```

```
# Consequence: If developers are not familiar with the @property decorator,
↪they might default to verbose getters/setters, leading to non-idiomatic
↪Python code.
```

50000

[7]:
```
#  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):
        self.__title = title
        self.__author = author
        self.__is_available = True

    def get_title(self):
        return self.__title

    def get_author(self):
        return self.__author

    def is_available(self):
        return self.__is_available

    def borrow_book(self):
        if self.__is_available:
            self.__is_available = False
            return f"You have successfully borrowed '{self.__title}'."
        else:
            return f"Sorry, '{self.__title}' is currently not available."

    def return_book(self):
        if not self.__is_available:
            self.__is_available = True
            return f"Thank you for returning '{self.__title}'."
        else:
            return f"'{self.__title}' is not borrowed, so it can't be returned."

book1 = Book("To Kill a Mockingbird", "Harper Lee")
book2 = Book("1984", "George Orwell")

print(book1.get_title(), "by", book1.get_author())
print("Available:", book1.is_available())

print(book1.borrow_book())
print("Available:", book1.is_available())

print(book1.return_book())
```

```python
print("Available:", book1.is_available())
```

```
To Kill a Mockingbird by Harper Lee
Available: True
You have successfully borrowed 'To Kill a Mockingbird'.
Available: False
Thank you for returning 'To Kill a Mockingbird'.
Available: True
```

[1]:
```python
# . Explain how encapsulation enhances code reusability and modularity in␣
↪Python programs.
class Book:
    def __init__(self, title, author):
        self.__title = title
        self.__author = author
        self.__is_available = True

    def borrow(self):
        if self.__is_available:
            self.__is_available = False
            return True
        return False

    def return_book(self):
        self.__is_available = True

class Member:
    def __init__(self, member_id, name):
        self.__member_id = member_id
        self.__name = name

    def get_member_id(self):
        return self.__member_id
class EBook(Book):
    def __init__(self, title, author, file_format):
        super().__init__(title, author)
        self.__file_format = file_format

    def get_format(self):
        return self.__file_format
```

[3]:
```python
# Describe the concept of information hiding in encapsulation. Why is it␣
↪essential in software development?
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance
```

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

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

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            raise ValueError("Invalid withdrawal amount.")
```

[1]:
```
# . 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_info):
        self.__name = name
        self.__address = address
        self.__contact_info = contact_info

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

    @name.setter
    def name(self, new_name):
        if isinstance(new_name, str) and new_name:
            self.__name = new_name
        else:
            raise ValueError("Name must be a non-empty string.")

    @property
    def address(self):
        return self.__address

    @address.setter
    def address(self, new_address):
        if isinstance(new_address, str) and new_address:
            self.__address = new_address
        else:
            raise ValueError("Address must be a non-empty string.")

    @property
```

```python
    def contact_info(self):
        return self.__contact_info

    @contact_info.setter
    def contact_info(self, new_contact_info):
        if isinstance(new_contact_info, str) and new_contact_info:
            self.__contact_info = new_contact_info
        else:
            raise ValueError("Contact information must be a non-empty string.")

    def __str__(self):
        return f"Customer(Name: {self.__name}, Address: {self.__address},␣
↪Contact: {self.__contact_info})"

if __name__ == "__main__":
    customer = Customer("John Doe", "123 Main St", "555-1234")
    print(customer)

    customer.name = "Jane Doe"
    customer.address = "456 Elm St"
    customer.contact_info = "555-5678"

    print(customer)
```

```
Customer(Name: John Doe, Address: 123 Main St, Contact: 555-1234)
Customer(Name: Jane Doe, Address: 456 Elm St, Contact: 555-5678)
```

[ ]:
```python
# Polymorphism :
```

[3]:
```python
# 1. What is polymorphism in Python? Explain how it is related to␣
↪object-oriented programming.
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def speak(self):
        return "Bark"

class Cat(Animal):
    def speak(self):
        return "Meow"

def animal_sound(animal):
    print(animal.speak())

dog = Dog()
```

```
cat = Cat()
animal_sound(dog)
animal_sound(cat)
```

```
Bark
Meow
```

[5]:
```
# 2. Describe the difference between compile-time polymorphism and runtime␣
↪polymorphism in Python.
class Example:
    def greet(self, name=None):
        if name:
            return f"Hello, {name}!"
        return "Hello!"

e = Example()
print(e.greet())
print(e.greet("Alice"))
```

```
Hello!
Hello, Alice!
```

[11]:
```
 # 3.Create a Python class hierarchy for shapes (e.g., circle, square,␣
↪triangle) and demonstrate polymorphismthrough a common method, such as␣
↪`calculate_area()`.
import math

class Shape:
    def calculate_area(self):
        raise NotImplementedError("Subclasses must implement this 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
```

```python
        self.height = height

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

def print_area(shape):
    print(f"The area of the shape is: {shape.calculate_area()}")

circle = Circle(5)
square = Square(4)
triangle = Triangle(3, 6)

print_area(circle)
print_area(square)
print_area(triangle)
```

```
The area of the shape is: 78.53981633974483
The area of the shape is: 16
The area of the shape is: 9.0
```

[13]:

```python
# 4. Explain the concept of method overriding in polymorphism. Provide an example.
class Animal:
    def speak(self):
        return "Some generic sound"

class Dog(Animal):
    def speak(self):
        return "Bark"

class Cat(Animal):
    def speak(self):
        return "Meow"

def animal_sound(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

animal_sound(dog)
animal_sound(cat)
```

```
Bark
Meow
```

[15]:

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

```
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this method.")

class Dog(Animal):
    def speak(self):
        return "Bark"

class Cat(Animal):
    def speak(self):
        return "Meow"

def animal_sound(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

animal_sound(dog)
animal_sound(cat)
```

```
Bark
Meow
```

[19]:

```
# .class Example:
def greet(self, name=None):
        if name:
            return f"Hello, {name}!"
        return "Hello!"

e = Example()

print(e.greet())         # Outputs: Hello!
print(e.greet("Alice"))  # Outputs: Hello, Alice!
```

```
Hello!
Hello, Alice!
```

[1]:

```
# 7. Discuss the use of abstract methods and classes in achieving polymorphism␣
 ↪in Python. Provide an example using the `abc` module.
from abc import ABC, abstractmethod
import math

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

```python
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

def print_area(shape: Shape):
    print(f"The area is: {shape.area()}")

circle = Circle(5)
rectangle = Rectangle(4, 6)

print_area(circle)
print_area(rectangle)
```

```
The area is: 78.53981633974483
The area is: 24
```

[3]:

```python
# 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.
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass

class Car(Vehicle):
    def start(self):
        print("The car's engine starts with a roar.")

class Bicycle(Vehicle):
    def start(self):
        print("The bicycle is ready to ride!")

class Boat(Vehicle):
```

```python
    def start(self):
        print("The boat's motor hums to life.")

def start_vehicle(vehicle: Vehicle):
    vehicle.start()

car = Car()
bicycle = Bicycle()
boat = Boat()

start_vehicle(car)
start_vehicle(bicycle)
start_vehicle(boat)
```

```
The car's engine starts with a roar.
The bicycle is ready to ride!
The boat's motor hums to life.
```

[5]:
```python
# 9. Explain the significance of the `isinstance()` and `issubclass()`␣
↪functions in Python polymorphism.
# isinstance()
def start_vehicle(vehicle):
    if isinstance(vehicle, Car):
        print("Starting a car...")
    elif isinstance(vehicle, Bicycle):
        print("Starting a bicycle...")
    else:
        print("Unknown vehicle type.")

start_vehicle(car)
start_vehicle(bicycle)


# issubclass()
def vehicle_info(vehicle_cls):
    if issubclass(vehicle_cls, Car):
        print("This is a car class.")
    elif issubclass(vehicle_cls, Bicycle):
        print("This is a bicycle class.")
    else:
        print("Unknown vehicle class.")

vehicle_info(Car)
vehicle_info(Bicycle)
```

```
Starting a car…
Starting a bicycle…
This is a car class.
```

This is a bicycle class.

[7]:
```
# 10.  What is the role of the `@abstractmethod` decorator in achieving␣
↪polymorphism in Python? Provide an example.
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

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

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

def animal_sound(animal: Animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

animal_sound(dog)
animal_sound(cat)
```

```
Woof!
Meow!
```

[9]:
```
# 11. . Create a Python class called `Shape` with a polymorphic method `area()`␣
↪that calculates the area of different shapes (e.g., circle, rectangle,␣
↪triangle)
from abc import ABC, abstractmethod
import math

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)
```

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

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

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

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

def print_area(shape: Shape):
    print(f"The area is: {shape.area()}")

circle = Circle(5)
rectangle = Rectangle(4, 6)
triangle = Triangle(3, 7)

print_area(circle)      # Output: The area is: 78.53981633974483
print_area(rectangle)   # Output: The area is: 24
print_area(triangle)    # Output: The area is: 10.5
```

```
The area is: 78.53981633974483
The area is: 24
The area is: 10.5
```

[ ]:
```python
# 12. Discuss the benefits of polymorphism in terms of code reusability and
 ↪flexibility in Python programs.
Code Reusability
Common Interface: By defining a common interface (e.g., through abstract base
 ↪classes or interface methods), different classes can be designed to fulfill
 ↪the same contract. This means you can write generic code that operates on
 ↪any subclass without knowing its specific type.

Reduced Redundancy: With polymorphism, you can avoid duplicating code.
 ↪Functions or methods can handle objects of different types using the same
 ↪code, which leads to cleaner and more maintainable code.

Example: In the Shape example, a single function (print_area()) can compute the
 ↪area for various shapes without needing separate implementations for each
 ↪shape type.
```

```
2. Flexibility
Dynamic Behavior: Polymorphism allows you to write code that can adapt to new␣
↪requirements without modifying existing code. You can introduce new classes␣
↪that extend functionality, and existing functions will work with these new␣
↪classes seamlessly.

Ease of Maintenance: When code needs to be updated or extended, polymorphism␣
↪makes it easier to incorporate changes. For example, if you want to add a␣
↪new shape (like Polygon), you just need to create a new class that␣
↪implements the area() method.

3. Interoperability
Unified Code Structure: Polymorphism enables different objects to be treated␣
↪similarly in a program. This interoperability simplifies the interaction␣
↪between components and makes it easier to integrate different systems or␣
↪modules.
4. Improved Code Organization
Clearer Design: Polymorphic designs often lead to better-organized code. By␣
↪following design principles like the Open/Closed Principle (classes should␣
↪be open for extension but closed for modification), developers can create␣
↪systems that are more intuitive and easier to navigate.
5. Simplified Code
Single Responsibility: Polymorphism allows classes to focus on their own␣
↪specific behavior while adhering to a common interface. This separation of␣
↪concerns makes each class simpler and easier to understand.
```

[11]:
```python
# 13.  Explain the use of the `super()` function in Python polymorphism. How␣
↪does it help call methods of parent classes?
class Shape:
    def area(self):
        return 0

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

    def area(self):
        parent_area = super().area()
        return parent_area + (3.14 * (self.radius ** 2))

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

    def area(self):
```

```
        parent_area = super().area()
        return parent_area + (self.width * self.height)

circle = Circle(5)
rectangle = Rectangle(4, 6)

print(f"Circle area: {circle.area()}")
print(f"Rectangle area: {rectangle.area()}")
```

```
Circle area: 78.5
Rectangle area: 24
```

[13]:

```
# 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.
from abc import ABC, abstractmethod

class BankAccount(ABC):
    def __init__(self, balance=0):
        self.balance = balance

    @abstractmethod
    def withdraw(self, amount):
        pass

class SavingsAccount(BankAccount):
    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds in Savings Account.")
        else:
            self.balance -= amount
            print(f"Withdrew {amount} from Savings Account. New balance: {self.
↪balance}")

class CheckingAccount(BankAccount):
    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds in Checking Account.")
        else:
            self.balance -= amount
            print(f"Withdrew {amount} from Checking Account. New balance: {self.
↪balance}")

class CreditCardAccount(BankAccount):
    def __init__(self, balance=0, credit_limit=1000):
        super().__init__(balance)
        self.credit_limit = credit_limit
```

```
    def withdraw(self, amount):
        if amount > (self.balance + self.credit_limit):
            print("Withdrawal exceeds credit limit in Credit Card Account.")
        else:
            self.balance -= amount
            print(f"Withdrew {amount} from Credit Card Account. New balance:␣
↪{self.balance}")

def perform_withdrawal(account: BankAccount, amount: float):
    account.withdraw(amount)

savings = SavingsAccount(500)
checking = CheckingAccount(300)
credit_card = CreditCardAccount(200)

perform_withdrawal(savings, 100)
perform_withdrawal(checking, 400)
perform_withdrawal(credit_card, 150)
perform_withdrawal(credit_card, 300)
```

```
Withdrew 100 from Savings Account. New balance: 400
Insufficient funds in Checking Account.
Withdrew 150 from Credit Card Account. New balance: 50
Withdrew 300 from Credit Card Account. New balance: -250
```

[15]:
```
# 15.  Describe the concept of operator overloading in Python and how it␣
↪relates to polymorphism. Provide examples using operators like `+` and `*`.
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        return NotImplemented

    def __mul__(self, scalar):
        if isinstance(scalar, (int, float)):
            return Vector(self.x * scalar, self.y * scalar)
        return NotImplemented

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
```

```python
v2 = Vector(5, 7)

v3 = v1 + v2
v4 = v1 * 3

print(v3)
print(v4)
```

```
Vector(7, 10)
Vector(6, 9)
```

[17]:

```python
# 16. What is dynamic polymorphism, and how is it achieved in Python?
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this method.")

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

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

def make_animal_speak(animal: Animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

make_animal_speak(dog)
make_animal_speak(cat)
```

```
Woof!
Meow!
```

[19]:

```python
#  Create a Python class hierarchy for employees in a company (e.g., manager,␣
↪developer, designer) and implement polymorphism through a common␣
↪`calculate_salary()` method.
from abc import ABC, abstractmethod

class Employee(ABC):
    def __init__(self, name, base_salary):
        self.name = name
        self.base_salary = base_salary

    @abstractmethod
    def calculate_salary(self):
```

```python
        pass

class Manager(Employee):
    def __init__(self, name, base_salary, bonus):
        super().__init__(name, base_salary)
        self.bonus = bonus

    def calculate_salary(self):
        return self.base_salary + self.bonus

class Developer(Employee):
    def __init__(self, name, base_salary, overtime_hours, overtime_rate):
        super().__init__(name, base_salary)
        self.overtime_hours = overtime_hours
        self.overtime_rate = overtime_rate

    def calculate_salary(self):
        return self.base_salary + (self.overtime_hours * self.overtime_rate)

class Designer(Employee):
    def __init__(self, name, base_salary, project_bonus):
        super().__init__(name, base_salary)
        self.project_bonus = project_bonus

    def calculate_salary(self):
        return self.base_salary + self.project_bonus

def print_salary(employee: Employee):
    print(f"{employee.name}'s salary: {employee.calculate_salary()}")

manager = Manager("Alice", 80000, 10000)
developer = Developer("Bob", 60000, 10, 50)
designer = Designer("Charlie", 50000, 5000)

print_salary(manager)
print_salary(developer)
print_salary(designer)
```

```
Alice's salary: 90000
Bob's salary: 60500
Charlie's salary: 55000
```

[21]:
```python
# 18.  Discuss the concept of function pointers and how they can be used to
↪achieve polymorphism in Python
def add(x, y):
    return x + y
```

```python
def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x / y if y != 0 else "Cannot divide by zero."

def perform_operation(operation, x, y):
    return operation(x, y)

x = 10
y = 5

print(perform_operation(add, x, y))
print(perform_operation(subtract, x, y))
print(perform_operation(multiply, x, y))
print(perform_operation(divide, x, y))
```

```
15
5
50
2.0
```

[1]:

```python
#  19.  Explain the role of interfaces and abstract classes in polymorphism,␣
 ↪drawing comparisons between them.
Abstract Classes
Definition: An abstract class is a class that cannot be instantiated on its own␣
 ↪and is designed to be a base class for other classes. It may contain both␣
 ↪abstract methods (methods without an implementation) and concrete methods␣
 ↪(methods with implementation).

Key Features:

Abstract Methods: Must be implemented by subclasses. They define a contract for␣
 ↪the subclasses, ensuring that certain methods are present.
Concrete Methods: Can provide default behavior that subclasses can inherit or␣
 ↪override.
Instantiation: Cannot create instances of an abstract class directly.
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass
```

```
    def sleep(self):
        return "Sleeping..."

class Dog(Animal):
    def sound(self):
        return "Woof!"

class Cat(Animal):
    def sound(self):
        return "Meow!"

Interfaces
Definition: An interface is a contract that defines a set of methods that a␣
↪class must implement. In Python, interfaces can be represented using␣
↪abstract base classes, but Python does not have a built-in keyword␣
↪specifically for interfaces as seen in languages like Java.

Key Features:

Method Signatures Only: Interfaces typically contain only method signatures␣
↪(abstract methods) without any implementation.
Multiple Inheritance: A class can implement multiple interfaces, allowing for␣
↪greater flexibility in design.
No State: Interfaces usually do not maintain state (attributes) and focus␣
↪solely on behavior.

class Vehicle(ABC):
    @abstractmethod
    def drive(self):
        pass

class Car(Vehicle):
    def drive(self):
        return "Driving a car."

class Bike(Vehicle):
    def drive(self):
        return "Riding a bike."
```

[3]:
```
# 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)
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def eat(self):
```

```python
        pass

    @abstractmethod
    def sleep(self):
        pass

    @abstractmethod
    def make_sound(self):
        pass

class Mammal(Animal):
    def eat(self):
        return "Eating grass or meat."

    def sleep(self):
        return "Sleeping in a den."

    @abstractmethod
    def make_sound(self):
        pass

class Bird(Animal):
    def eat(self):
        return "Eating seeds or insects."

    def sleep(self):
        return "Sleeping in a nest."

    @abstractmethod
    def make_sound(self):
        pass

class Reptile(Animal):
    def eat(self):
        return "Eating insects or small animals."

    def sleep(self):
        return "Sleeping in a sun spot."

    @abstractmethod
    def make_sound(self):
        pass

class Lion(Mammal):
    def make_sound(self):
        return "Roar!"
```

```python
class Elephant(Mammal):
    def make_sound(self):
        return "Trumpet!"

class Parrot(Bird):
    def make_sound(self):
        return "Squawk!"

class Sparrow(Bird):
    def make_sound(self):
        return "Chirp!"

# Specific Reptiles
class Snake(Reptile):
    def make_sound(self):
        return "Hiss!"

class Lizard(Reptile):
    def make_sound(self):
        return "Sssss!"

def interact_with_animal(animal: Animal):
    print(f"{animal.__class__.__name__}:")
    print(f" - Eating: {animal.eat()}")
    print(f" - Sleeping: {animal.sleep()}")
    print(f" - Sound: {animal.make_sound()}")
    print()

lion = Lion()
elephant = Elephant()
parrot = Parrot()
sparrow = Sparrow()
snake = Snake()
lizard = Lizard()

animals = [lion, elephant, parrot, sparrow, snake, lizard]
for animal in animals:
    interact_with_animal(animal)
```

```
Lion:
 - Eating: Eating grass or meat.
 - Sleeping: Sleeping in a den.
 - Sound: Roar!

Elephant:
 - Eating: Eating grass or meat.
 - Sleeping: Sleeping in a den.
 - Sound: Trumpet!
```

```
Parrot:
 - Eating: Eating seeds or insects.
 - Sleeping: Sleeping in a nest.
 - Sound: Squawk!

Sparrow:
 - Eating: Eating seeds or insects.
 - Sleeping: Sleeping in a nest.
 - Sound: Chirp!

Snake:
 - Eating: Eating insects or small animals.
 - Sleeping: Sleeping in a sun spot.
 - Sound: Hiss!

Lizard:
 - Eating: Eating insects or small animals.
 - Sleeping: Sleeping in a sun spot.
 - Sound: Sssss!
```

[ ]:
```python
# Abstraction:
```

[5]:
```python
# 1.  What is abstraction in Python, and how does it relate to object-oriented␣
 ↪programming
from abc import ABC, abstractmethod

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

    @abstractmethod
    def perimeter(self):
        pass

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

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

    def perimeter(self):
        return 2 * (self.width + self.height)
```

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

    def area(self):
        return 3.14 * self.radius * self.radius

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

def display_shape_info(shape: Shape):
    print(f"Area: {shape.area()}")
    print(f"Perimeter: {shape.perimeter()}")

rectangle = Rectangle(5, 10)
circle = Circle(7)

display_shape_info(rectangle)
display_shape_info(circle)
```

```
Area: 50
Perimeter: 30
Area: 153.86
Perimeter: 43.96
```

[ ]:
```
# 2. Describe the benefits of abstraction in terms of code organization and
 ↪complexity reduction.
Benefits of Abstraction
Simplified Interface:

User-Friendly Interaction: Abstraction allows developers to interact with
 ↪complex systems through simplified interfaces. Users can utilize methods
 ↪without needing to understand the underlying implementation details, making
 ↪it easier to work with the code.
Focus on High-Level Logic: Developers can concentrate on what the code does
 ↪rather than how it does it, leading to more efficient problem-solving and
 ↪code management.
Improved Code Organization:

Separation of Concerns: Abstraction encourages a clear separation between the
 ↪interface and implementation. This modular approach helps in organizing code
 ↪better, allowing different teams to work on various components without
 ↪interfering with each other.
Clear Structure: By defining abstract classes and interfaces, the relationships
 ↪between components become clearer, improving the overall architecture of the
 ↪software.
```

```
Reduced Complexity:

Hiding Implementation Details: Abstraction allows developers to hide the␣
↪complexities of the underlying system, exposing only essential features.␣
↪This makes it easier to manage and understand the system as a whole.
Less Cognitive Load: By reducing the amount of information a developer needs to␣
↪process at any given time, abstraction lowers the cognitive load, making it␣
↪easier to think through problems and implement solutions.
Enhanced Maintainability:

Easier Modifications: Changes to implementation details can often be made␣
↪without affecting the code that relies on the abstract interface. This␣
↪encapsulation leads to easier maintenance and updates.
Reduced Risk of Errors: By working with higher-level abstractions, developers␣
↪are less likely to introduce errors related to low-level details, as the␣
↪complex implementation is managed within the abstract classes.
Increased Reusability:

Code Reusability: Abstract classes and interfaces can be reused across␣
↪different projects or within the same project. This promotes DRY (Don't␣
↪Repeat Yourself) principles, reducing code duplication and improving␣
↪consistency.
Flexible Extensibility: New features or variations can be added by creating new␣
↪subclasses or implementing new interfaces without altering existing code,␣
↪enhancing the system’s extensibility.
Facilitated Testing:

Mocking and Stubbing: In testing environments, abstraction allows for easier␣
↪creation of mock objects that adhere to the same interface, enabling␣
↪developers to test components in isolation.
Focused Testing: Since implementation details are hidden, tests can focus on␣
↪the functionality exposed through the abstract interfaces, making it clearer␣
↪what needs to be tested.
```

[7]:
```
#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
from abc import ABC, abstractmethod
import math

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

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

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

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

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

def display_area(shape: Shape):
    print(f"The area of the {shape.__class__.__name__} is: {shape.
 ↪calculate_area():.2f}")

circle = Circle(5)
rectangle = Rectangle(4, 6)

display_area(circle)
display_area(rectangle)
```

```
The area of the Circle is: 78.54
The area of the Rectangle is: 24.00
```

[9]:

```
# 4. Explain the concept of abstract classes in Python and how they are defined␣
 ↪using the `abc` module. Provide an example.
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

def animal_sound(animal: Animal):
    print(f"The animal makes a sound: {animal.make_sound()}")
```

```
dog = Dog()
cat = Cat()

animal_sound(dog)
animal_sound(cat)
```

```
The animal makes a sound: Woof!
The animal makes a sound: Meow!
```

[ ]:
```
# 5. How do abstract classes differ from regular classes in Python? Discuss
  ↪their use cases.
Differences Between Abstract Classes and Regular Classes
Instantiation:

Abstract Classes: Cannot be instantiated directly. You must subclass an
  ↪abstract class and implement its abstract methods in order to create
  ↪instances of that subclass.
Regular Classes: Can be instantiated directly, allowing you to create objects
  ↪from the class.
Abstract Methods:

Abstract Classes: Can contain one or more abstract methods (methods defined
  ↪with @abstractmethod decorator). Subclasses are required to provide
  ↪implementations for these methods.
Regular Classes: Do not have abstract methods. All methods can have
  ↪implementations, and subclasses can override them if needed.
Purpose:

Abstract Classes: Serve as a blueprint for other classes, defining a common
  ↪interface and ensuring that derived classes implement certain methods. They
  ↪promote code organization and enforce consistency across related classes.
Regular Classes: Implement specific functionality and behavior. They can be
  ↪used independently and do not enforce a common interface for subclasses.
Inheritance:

Abstract Classes: Are primarily used for inheritance and defining relationships
  ↪between classes. They help in creating a hierarchy of classes that share
  ↪common functionality.
Regular Classes: Can be used standalone or as part of a class hierarchy. They
  ↪may not necessarily have relationships with other classes.
Use Cases for Abstract Classes
Frameworks and Libraries:
```

```
Abstract classes are often used in frameworks where you define a set of␣
↪behaviors that various components should implement. For example, a GUI␣
↪framework might define an abstract class Widget with methods like draw() and␣
↪resize(), requiring all widgets (buttons, sliders, etc.) to implement these␣
↪methods.
Plugin Systems:

In systems that support plugins, an abstract class can define the expected␣
↪interface for plugins. Each plugin would then inherit from the abstract␣
↪class and provide its own implementation of the required methods.
Data Processing:

When designing systems for data processing (e.g., different types of data␣
↪sources), you can create an abstract class DataSource with methods like␣
↪read_data() and write_data(). Specific implementations can be created for␣
↪reading from a database, file, or API.
Game Development:

Abstract classes can define a base class for game objects, such as GameObject,␣
↪with methods like update() and render(). Different game entities (e.g.,␣
↪player, enemies, NPCs) would inherit from this class and implement their␣
↪specific behavior.
```

[11]:

```
# 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.
class BankAccount:
    def __init__(self, account_number, initial_balance=0):
        self.account_number = account_number
        self._balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            print(f"Deposited: ${amount:.2f}. New balance: ${self._balance:.2f}.
↪")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount > 0:
            if amount <= self._balance:
                self._balance -= amount
                print(f"Withdrew: ${amount:.2f}. New balance: ${self._balance:.
↪2f}.")
            else:
                print("Insufficient funds.")
```

```python
        else:
            print("Withdrawal amount must be positive.")

    def get_balance(self):
        return self._balance

if __name__ == "__main__":
    account = BankAccount("12345678", initial_balance=100.00)

    account.deposit(50)
    account.withdraw(30)
    account.withdraw(200)

    print(f"Current balance (accessed via method): ${account.get_balance():.
 ↪2f}")
```

```
Deposited: $50.00. New balance: $150.00.
Withdrew: $30.00. New balance: $120.00.
Insufficient funds.
Current balance (accessed via method): $120.00
```

[13]:
```python
# Discuss the concept of interface classes in Python and their role in␣
 ↪achieving abstraction.
from abc import ABC, abstractmethod

class Drawable(ABC):
    @abstractmethod
    def draw(self):
        pass

class Circle(Drawable):
    def draw(self):
        return "Drawing a Circle"

class Rectangle(Drawable):
    def draw(self):
        return "Drawing a Rectangle"

def render_shape(shape: Drawable):
    print(shape.draw())

circle = Circle()
rectangle = Rectangle()

render_shape(circle)
render_shape(rectangle)
```

```
Drawing a Circle
```

Drawing a Rectangle

[15]:
```python
# 8. Create a Python class hierarchy for animals and implement abstraction by defining common methods (e.g., `eat()`, `sleep()`) in an abstract base class.
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def eat(self):
        pass

    @abstractmethod
    def sleep(self):
        pass

class Dog(Animal):
    def eat(self):
        return "The dog is eating dog food."

    def sleep(self):
        return "The dog is sleeping in its kennel."

class Cat(Animal):
    def eat(self):
        return "The cat is eating cat food."

    def sleep(self):
        return "The cat is sleeping on the couch."

def animal_behavior(animal: Animal):
    print(animal.eat())
    print(animal.sleep())

if __name__ == "__main__":
    dog = Dog()
    cat = Cat()

    print("Dog Behavior:")
    animal_behavior(dog)

    print("\nCat Behavior:")
    animal_behavior(cat)
```

```
Dog Behavior:
The dog is eating dog food.
The dog is sleeping in its kennel.

Cat Behavior:
```

```
The cat is eating cat food.
The cat is sleeping on the couch.
```

[17]:
```python
# 9.  Explain the significance of encapsulation in achieving abstraction.␣
↪Provide examples.
class BankAccount:
    def __init__(self, account_number, initial_balance=0):
        self.account_number = account_number
        self.__balance = initial_balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited: ${amount:.2f}. New balance: ${self.__balance:.
↪2f}.")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount > 0:
            if amount <= self.__balance:
                self.__balance -= amount
                print(f"Withdrew: ${amount:.2f}. New balance: ${self.__balance:.
↪2f}.")
            else:
                print("Insufficient funds.")
        else:
            print("Withdrawal amount must be positive.")

    def get_balance(self):
        return self.__balance

if __name__ == "__main__":
    account = BankAccount("12345678", initial_balance=100.00)

    account.deposit(50)
    account.withdraw(30)
    print(f"Current balance: ${account.get_balance():.2f}")
```

```
Deposited: $50.00. New balance: $150.00.
Withdrew: $30.00. New balance: $120.00.
Current balance: $120.00
```

[19]:
```python
# 10.  What is the purpose of abstract methods, and how do they enforce␣
↪abstraction in Python classes?
from abc import ABC, abstractmethod

class Shape(ABC):
```

```python
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

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

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

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

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

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

    def perimeter(self):
        return 2 * (self.width + self.height)

if __name__ == "__main__":
    circle = Circle(5)
    rectangle = Rectangle(4, 6)

    print(f"Circle area: {circle.area()}")
    print(f"Rectangle area: {rectangle.area()}")
```

```
Circle area: 78.5
Rectangle area: 24
```

[21]:

```python
# 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.
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start(self):
```

```python
        pass

    @abstractmethod
    def stop(self):
        pass

class Car(Vehicle):
    def start(self):
        return "Car is starting with a roar!"

    def stop(self):
        return "Car is stopping smoothly."

class Bicycle(Vehicle):
    def start(self):
        return "Bicycle is getting ready to ride!"

    def stop(self):
        return "Bicycle is coming to a halt."

def vehicle_action(vehicle: Vehicle):
    print(vehicle.start())
    print(vehicle.stop())

if __name__ == "__main__":
    car = Car()
    bicycle = Bicycle()

    print("Car Actions:")
    vehicle_action(car)

    print("\nBicycle Actions:")
    vehicle_action(bicycle)
```

```
Car Actions:
Car is starting with a roar!
Car is stopping smoothly.

Bicycle Actions:
Bicycle is getting ready to ride!
Bicycle is coming to a halt.
```

[23]:

```python
#12. . Describe the use of abstract properties in Python and how they can be␣
 ↪employed in abstract classes.
from abc import ABC, abstractmethod

class Shape(ABC):
```

```python
    @property
    @abstractmethod
    def area(self):
        """Return the area of the shape."""
        pass

    @property
    @abstractmethod
    def perimeter(self):
        """Return the perimeter of the shape."""
        pass

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

    @property
    def area(self):
        return 3.14 * (self._radius ** 2)

    @property
    def perimeter(self):
        return 2 * 3.14 * self._radius

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

    @property
    def area(self):
        return self._width * self._height

    @property
    def perimeter(self):
        return 2 * (self._width + self._height)

if __name__ == "__main__":
    circle = Circle(5)
    rectangle = Rectangle(4, 6)

    print(f"Circle Area: {circle.area}")
    print(f"Circle Perimeter: {circle.perimeter}")

    print(f"Rectangle Area: {rectangle.area}")
    print(f"Rectangle Perimeter: {rectangle.perimeter}")
```

```
Circle Area: 78.5
```

```
Circle Perimeter: 31.400000000000002
Rectangle Area: 24
Rectangle Perimeter: 20
```

[25]:

```python
# 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.
from abc import ABC, abstractmethod

class Employee(ABC):
    @abstractmethod
    def get_salary(self):
        pass

class Manager(Employee):
    def __init__(self, name, base_salary, bonus):
        self.name = name
        self.base_salary = base_salary
        self.bonus = bonus

    def get_salary(self):
        return self.base_salary + self.bonus

class Developer(Employee):
    def __init__(self, name, base_salary, overtime_hours, overtime_rate):
        self.name = name
        self.base_salary = base_salary
        self.overtime_hours = overtime_hours
        self.overtime_rate = overtime_rate

    def get_salary(self):
        overtime_pay = self.overtime_hours * self.overtime_rate
        return self.base_salary + overtime_pay

class Designer(Employee):
    def __init__(self, name, base_salary):
        self.name = name
        self.base_salary = base_salary

    def get_salary(self):
        return self.base_salary

def display_salary(employee: Employee):
    print(f"{employee.name}'s salary: ${employee.get_salary():.2f}")

if __name__ == "__main__":
    manager = Manager("Alice", 80000, 10000)
```

```
    developer = Developer("Bob", 60000, 10, 50)
    designer = Designer("Charlie", 50000)

    display_salary(manager)
    display_salary(developer)
    display_salary(designer)
```

```
Alice's salary: $90000.00
Bob's salary: $60500.00
Charlie's salary: $50000.00
```

[ ]:
```
# 14. . Discuss the differences between abstract classes and concrete classes␣
↪in Python, including their instantiation.
Abstract Classes
Definition:

An abstract class is a class that cannot be instantiated on its own. It is␣
↪designed to be a blueprint for other classes. It may contain abstract␣
↪methods (methods without implementation) that must be implemented by␣
↪subclasses.
Purpose:

The primary purpose of an abstract class is to define a common interface and to␣
↪provide a foundation for other classes. It enforces that subclasses␣
↪implement specific methods, promoting a consistent structure across␣
↪different implementations.
Instantiation:

Abstract classes cannot be instantiated directly. Attempting to create an␣
↪instance of an abstract class will result in a TypeError. This is to prevent␣
↪the creation of incomplete objects that do not implement all required␣
↪methods.


Concrete Classes
Definition:

A concrete class is a class that can be instantiated. It provides complete␣
↪implementations of all its methods, including any methods defined in its␣
↪abstract base classes.
Purpose:

The purpose of a concrete class is to provide specific implementations of the␣
↪methods defined in the abstract class. It represents a fully functional␣
↪object that can be used in programs.
Instantiation:

```

```
Concrete classes can be instantiated. You can create objects of a concrete
class, and these objects can be used to access the class's methods and
properties.
```

[27]:
```
# 15.  Explain the concept of abstract data types (ADTs) and their role in
achieving abstraction in Python
'''Key Concepts of Abstract Data Types (ADTs)
Definition:

An Abstract Data Type is a model for data structures that defines the data type
in terms of its behavior (operations) from the point of view of a user. ADTs
are characterized by a set of values and a set of operations that can be
performed on those values.
Operations:

ADTs specify the operations that can be performed, such as insertion, deletion,
searching, and accessing elements. These operations define the interface of
the ADT but do not specify the implementation details.
Encapsulation:

ADTs encapsulate data and expose only the operations needed to interact with
that data. This encapsulation hides the underlying implementation, allowing
users to work with the data type without needing to understand its
complexities.
Examples of ADTs:

Common examples of ADTs include:
Stack: A collection of elements with operations like push, pop, and peek.
Queue: A collection of elements with operations like enqueue, dequeue, and
front.
List: A collection with operations like append, remove, and get.
Dictionary: A collection of key-value pairs with operations like insert,
delete, and lookup.
Role of ADTs in Achieving Abstraction in Python
Separation of Interface and Implementation:

ADTs promote a clear separation between what operations are available
(interface) and how those operations are implemented (implementation). This
allows developers to change the implementation without affecting code that
relies on the ADT.
Improved Code Organization:

By defining data types as ADTs, you can create organized and modular code. This
helps in managing complexity, as users interact with well-defined operations
rather than dealing with low-level details.
```

```
Flexibility and Maintainability:

ADTs enhance flexibility because the implementation can be changed (e.g., using
 a different algorithm or data structure) without affecting the code that
 uses the ADT. This makes the code easier to maintain and evolve.
Facilitates Collaboration:

In collaborative environments, ADTs allow different team members to work on
 different parts of a system simultaneously. For example, one developer can
 work on the ADT's interface while another focuses on the implementation.'''

class Stack:
    def __init__(self):
        self._items = []  # Internal representation (encapsulated)

    def is_empty(self):
        return len(self._items) == 0

    def push(self, item):
        self._items.append(item)

    def pop(self):
        if not self.is_empty():
            return self._items.pop()
        raise IndexError("Pop from empty stack")

    def peek(self):
        if not self.is_empty():
            return self._items[-1]
        raise IndexError("Peek from empty stack")

    def size(self):
        return len(self._items)

if __name__ == "__main__":
    stack = Stack()
    stack.push(1)
    stack.push(2)
    print(stack.peek())  # Output: 2
    print(stack.pop())   # Output: 2
    print(stack.size())  # Output: 1
```

```
2
2
1
```

[29]:

```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.
from abc import ABC, abstractmethod

class Computer(ABC):
    @abstractmethod
    def power_on(self):
        pass

    @abstractmethod
    def shutdown(self):
        pass

class Laptop(Computer):
    def power_on(self):
        return "Laptop is powering on."

    def shutdown(self):
        return "Laptop is shutting down."

class Desktop(Computer):
    def power_on(self):
        return "Desktop is powering on."

    def shutdown(self):
        return "Desktop is shutting down."

def operate_computer(computer: Computer):
    print(computer.power_on())
    print(computer.shutdown())

if __name__ == "__main__":
    laptop = Laptop()
    desktop = Desktop()

    print("Laptop Operations:")
    operate_computer(laptop)

    print("\nDesktop Operations:")
    operate_computer(desktop)
```

```
Laptop Operations:
Laptop is powering on.
```

```
Laptop is shutting down.

Desktop Operations:
Desktop is powering on.
Desktop is shutting down.
```

[ ]:
```
# 17.  . Discuss the benefits of using abstraction in large-scale software␣
↪development projects.
Abstraction is a fundamental principle in software engineering, particularly␣
↪beneficial in large-scale software development projects. Here are several␣
↪key benefits of using abstraction:

1. Simplification of Complex Systems
Managing Complexity: Abstraction helps break down complex systems into␣
↪manageable components. By hiding the intricate details of implementation,␣
↪developers can focus on higher-level functionalities, making it easier to␣
↪understand and work with the system as a whole.
Clearer Interfaces: With abstraction, components expose only the necessary␣
↪methods and properties, providing clear interfaces. This simplifies␣
↪interactions and reduces the cognitive load on developers.
2. Enhanced Modularity
Independent Development: By defining clear interfaces and abstract classes,␣
↪different teams can work on separate components without interfering with␣
↪each other. This promotes parallel development, increasing productivity.
Ease of Testing: Modular components can be tested independently, allowing for␣
↪more straightforward debugging and validation of functionality.
3. Improved Code Reusability
Reusable Components: Abstract classes and interfaces can be reused across␣
↪different projects or within the same project. This reduces code duplication␣
↪and encourages the use of well-tested components.
Easier Refactoring: When changes are needed, abstraction allows developers to␣
↪modify implementations without affecting other parts of the code that rely␣
↪on the abstracted interfaces.
4. Flexibility and Extensibility
Adapting to Change: As project requirements evolve, abstract components can be␣
↪extended or modified with minimal impact on the overall system. This␣
↪adaptability is crucial in large projects where requirements may change␣
↪frequently.
Support for Multiple Implementations: Abstraction allows for multiple concrete␣
↪implementations of an interface. This means that different algorithms or␣
↪data structures can be used interchangeably without changing the client code.
5. Encapsulation of Implementation Details
Hiding Complexity: Abstraction encapsulates the details of how operations are␣
↪performed, allowing developers to use components without needing to␣
↪understand their internal workings. This is particularly useful for complex␣
↪operations.
```

```
Security and Integrity: By restricting access to certain methods and
properties, abstraction can help protect the integrity of the data and
enforce rules about how it can be modified.
6. Facilitating Collaboration
Clear Responsibilities: With well-defined abstractions, team members can
clearly understand their responsibilities and how their work fits into the
larger system. This clarity fosters better collaboration among team members.
Documentation and Onboarding: Abstraction provides a natural documentation
mechanism, where the interfaces serve as a guide for how to interact with
components, making it easier for new developers to onboard.
7. Consistent Design Patterns
Standardization: Abstraction encourages the use of design patterns, which are
proven solutions to common problems in software design. This consistency
helps maintain a standard approach to development across the project.
```

[31]:
```
# 18. Explain how abstraction enhances code reusability and modularity in
Python programs.
'''1. Separation of Interface and Implementation
Clear Interfaces: Abstraction allows you to define a clear interface for
classes or modules. By specifying what operations can be performed without
detailing how they are implemented, developers can use these interfaces
without needing to understand the underlying code.
Loose Coupling: This separation reduces dependencies between different parts of
a program. Changes in implementation can occur without affecting code that
relies on the abstracted interface, promoting better modularity.
2. Modular Design
Encapsulation: Abstraction promotes encapsulation, where data and behavior are
bundled together while hiding the implementation details. This encapsulation
leads to well-defined modules, each responsible for specific functionalities.
Independent Development: Different modules can be developed and tested
independently. Teams can work on separate components concurrently,
facilitating parallel development and speeding up the overall development
process.
3. Code Reusability
Reusable Abstract Components: Abstract classes and interfaces can be reused
across multiple projects or within different parts of the same project. For
example, a generic Shape interface can be implemented by multiple classes
like Circle, Rectangle, and Triangle, all sharing the same interface while
providing their unique implementations.
Flexible Implementations: Developers can create new implementations that
conform to existing interfaces without changing the clients that use those
interfaces. This allows for easy swapping of implementations (e.g.,
switching between different sorting algorithms) without altering the calling
code.
4. Easier Maintenance and Refactoring
```

```
Simplified Changes: When an implementation needs to be changed or improved,␣
 ↪abstraction allows developers to modify only the concrete class without␣
 ↪affecting other parts of the code. For instance, if a new algorithm is␣
 ↪needed for a data processing task, only the implementation class needs to be␣
 ↪updated.
Consistent Updates: As new features are added or requirements change, existing␣
 ↪abstract interfaces can be extended, allowing for backward compatibility and␣
 ↪reducing the need for widespread changes in the codebase.
5. Improved Readability and Understanding
Self-Documenting Code: Well-defined abstractions make the code more intuitive␣
 ↪and easier to read. When developers see a class that implements an abstract␣
 ↪interface, they understand what functionalities to expect without needing to␣
 ↪delve into the implementation details.
Guided Usage: Documentation associated with abstract classes and interfaces␣
 ↪serves as a guide for developers on how to use them, which can streamline␣
 ↪onboarding and collaboration.'''
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass

    @abstractmethod
    def stop(self):
        pass

class Car(Vehicle):
    def start(self):
        return "Car is starting."

    def stop(self):
        return "Car is stopping."

class Bike(Vehicle):
    def start(self):
        return "Bike is starting."

    def stop(self):
        return "Bike is stopping."

def operate_vehicle(vehicle: Vehicle):
    print(vehicle.start())
    print(vehicle.stop())

if __name__ == "__main__":
    car = Car()
```

```
    bike = Bike()

    operate_vehicle(car)
    operate_vehicle(bike)
```

```
Car is starting.
Car is stopping.
Bike is starting.
Bike is stopping.
```

[33]:

```
# 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.
from abc import ABC, abstractmethod

class Library(ABC):
    @abstractmethod
    def add_book(self, title: str, author: str):
        """Add a book to the library."""
        pass

    @abstractmethod
    def borrow_book(self, title: str, member_name: str):
        """Borrow a book from the library."""
        pass

class PublicLibrary(Library):
    def __init__(self):
        self.books = {}

    def add_book(self, title: str, author: str):
        self.books[title] = {"author": author, "available": True}
        print(f"Added book: '{title}' by {author}.")

    def borrow_book(self, title: str, member_name: str):
        if title in self.books and self.books[title]["available"]:
            self.books[title]["available"] = False
            print(f"{member_name} borrowed '{title}'.")
        else:
            print(f"'{title}' is not available for borrowing.")

class UniversityLibrary(Library):
    def __init__(self):
        self.books = {}

    def add_book(self, title: str, author: str):
        self.books[title] = {"author": author, "available": True}
```

```python
        print(f"Added book: '{title}' by {author}.")

    def borrow_book(self, title: str, member_name: str):
        if title in self.books and self.books[title]["available"]:
            self.books[title]["available"] = False
            print(f"{member_name} borrowed '{title}' from the university␣
↪library.")
        else:
            print(f"'{title}' is not available for borrowing from the␣
↪university library.")

def operate_library(library: Library):
    library.add_book("1984", "George Orwell")
    library.add_book("To Kill a Mockingbird", "Harper Lee")
    library.borrow_book("1984", "Alice")
    library.borrow_book("The Great Gatsby", "Bob")

if __name__ == "__main__":
    public_library = PublicLibrary()
    university_library = UniversityLibrary()

    print("Public Library Operations:")
    operate_library(public_library)

    print("\nUniversity Library Operations:")
    operate_library(university_library)
```

```
Public Library Operations:
Added book: '1984' by George Orwell.
Added book: 'To Kill a Mockingbird' by Harper Lee.
Alice borrowed '1984'.
'The Great Gatsby' is not available for borrowing.

University Library Operations:
Added book: '1984' by George Orwell.
Added book: 'To Kill a Mockingbird' by Harper Lee.
Alice borrowed '1984' from the university library.
'The Great Gatsby' is not available for borrowing from the university library.
```

[35]:
```python
# 20. Describe the concept of method abstraction in Python and how it relates␣
↪to polymorphism.
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass
```

```python
class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

def animal_sound(animal: Animal):
    print(animal.sound())

if __name__ == "__main__":
    dog = Dog()
    cat = Cat()

    animal_sound(dog)
    animal_sound(cat)
```

```
Bark
Meow
```

[ ]:
```python
# Composition:
```

[37]:
```python
# 1. Explain the concept of composition in Python and how it is used to build␣
 ↪complex objects from simpler ones
class Engine:
    def start(self):
        return "Engine starting..."

    def stop(self):
        return "Engine stopping..."

class Wheels:
    def rotate(self):
        return "Wheels are rotating..."

class Car:
    def __init__(self):
        self.engine = Engine()
        self.wheels = Wheels()

    def start(self):
        return self.engine.start() + " Car is ready to go!"

    def stop(self):
        return self.engine.stop() + " Car has stopped."
```

```python
    def drive(self):
        return self.wheels.rotate() + " Car is moving."

if __name__ == "__main__":
    my_car = Car()
    print(my_car.start())
    print(my_car.drive())
    print(my_car.stop())
```

```
Engine starting… Car is ready to go!
Wheels are rotating… Car is moving.
Engine stopping… Car has stopped.
```

[ ]:
```
# 2.  Describe the difference between composition and inheritance in
 ↪object-oriented programming.
1. Definition
Inheritance:

Inheritance is a mechanism where a new class (subclass) derives from an
 ↪existing class (superclass). The subclass inherits attributes and methods
 ↪from the superclass, allowing for code reuse and the extension of existing
 ↪functionality.
"Is-a" Relationship: Inheritance represents an "is-a" relationship (e.g., a Dog
 ↪is an Animal).
Composition:

Composition involves constructing a class using instances of other classes,
 ↪effectively building complex objects from simpler ones. It allows one class
 ↪to contain references to other classes as its components.
"Has-a" Relationship: Composition represents a "has-a" relationship (e.g., a
 ↪Car has an Engine).
2. Flexibility
Inheritance:

Inheritance can lead to a rigid hierarchy where changes in the superclass can
 ↪inadvertently affect all subclasses. This can make the codebase more
 ↪difficult to manage and maintain.
It’s harder to modify behavior at runtime since the relationship is fixed.
Composition:

Composition offers greater flexibility, allowing you to change components at
 ↪runtime. You can easily swap out parts or add new functionality by changing
 ↪the composition of objects without altering existing code.
This flexibility makes it easier to adapt to changing requirements.
3. Encapsulation
Inheritance:
```

```
Inheritance exposes the implementation details of the superclass to its
 ↪subclasses, which can lead to tighter coupling between classes.
Subclasses are often dependent on the specifics of the superclass, making it
 ↪harder to change the implementation without affecting the subclasses.
Composition:

Composition promotes encapsulation by allowing each component to manage its
 ↪internal state and behavior independently. The composite class interacts
 ↪with these components through defined interfaces, reducing dependencies.
This separation helps in maintaining the integrity of individual components.
4. Reusability
Inheritance:

Code reuse occurs through the hierarchy established by inheritance. However,
 ↪this can sometimes lead to code duplication if multiple subclasses need
 ↪similar behavior that is not adequately captured in the superclass.
Composition:

Composition encourages higher reusability of components since classes can be
 ↪composed in various ways to create new behaviors without altering the
 ↪components themselves.
Components can be reused across different classes without being tied to a
 ↪specific inheritance hierarchy.
5. Complexity
Inheritance:

While inheritance can simplify some relationships by creating a clear
 ↪hierarchy, it can also lead to complexity, particularly with deep
 ↪inheritance trees (the "inheritance hell" problem).
Managing multiple levels of inheritance can become complicated and less
 ↪intuitive.
Composition:

Composition can lead to a more complex initial design as it requires careful
 ↪planning of how objects interact. However, this complexity often results in
 ↪a more manageable and adaptable system in the long run.
6. Use Cases
Inheritance:

Best used when there is a clear hierarchical relationship and when subclasses
 ↪share a significant amount of behavior and attributes.
Common in scenarios where polymorphism is desired, allowing for the
 ↪substitution of subclasses for their superclasses.
Composition:
```

```
Preferred in situations where behaviors can be shared and reused across␣
↪different classes without the constraints of a hierarchy.
Ideal for building systems that require more flexibility and adaptability, such␣
↪as frameworks or systems that need to evolve over time.
```

[39]:

```
# 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.
from datetime import datetime

class Author:
    def __init__(self, name: str, birthdate: str):
        self.name = name
        self.birthdate = datetime.strptime(birthdate, "%Y-%m-%d")

    def __str__(self):
        return f"{self.name}, born on {self.birthdate.strftime('%B %d, %Y')}"

class Book:
    def __init__(self, title: str, author: Author, publication_year: int):
        self.title = title
        self.author = author
        self.publication_year = publication_year

    def __str__(self):
        return f"'{self.title}' by {self.author} (Published in {self.
↪publication_year})"

if __name__ == "__main__":
    author = Author("George Orwell", "1903-06-25")
    book = Book("1984", author, 1949)

    print(book)
```

```
'1984' by George Orwell, born on June 25, 1903 (Published in 1949)
```

[ ]:

```
# 4.  Discuss the benefits of using composition over inheritance in Python,␣
↪especially in terms of code flexibility and reusability.
1. Flexibility
Dynamic Behavior:

Composition allows you to change the behavior of objects at runtime by swapping␣
↪out components. For example, you can replace a Car's engine with a different␣
↪type without modifying the Car class itself. This flexibility is harder to␣
↪achieve with inheritance, where relationships are typically fixed.
Easier Maintenance:
```

```
Since components are loosely coupled, you can modify or replace them without␣
    ↪affecting other parts of the system. This makes maintaining and updating the␣
    ↪codebase simpler, especially in large applications.
Avoiding Inheritance Hierarchies:

Complex inheritance hierarchies can lead to difficulties in managing and␣
    ↪understanding relationships between classes. Composition allows for a more␣
    ↪modular approach where behavior can be constructed as needed, avoiding the␣
    ↪pitfalls of deep inheritance.
2. Code Reusability
Reuse Components:

Composition enables the reuse of existing classes as components in new contexts.
    ↪ For example, if you have a Database class, you can use it in multiple␣
    ↪classes (like User, Order, etc.) without creating a new hierarchy. This␣
    ↪promotes DRY (Don't Repeat Yourself) principles.
Decoupling:

When classes are designed using composition, they are generally more␣
    ↪independent of each other. This decoupling makes it easier to reuse␣
    ↪components across different parts of an application or even in different␣
    ↪projects.
3. Improved Design
Single Responsibility Principle:

Composition promotes the design of smaller, more focused classes that adhere to␣
    ↪the Single Responsibility Principle. Each class can handle a specific piece␣
    ↪of functionality, making it easier to manage and understand.
Clearer Interfaces:

When using composition, the responsibilities of each component are clearly␣
    ↪defined, leading to clearer interfaces. This can enhance collaboration among␣
    ↪team members as the design and purpose of each class become more apparent.
4. Simplifying Testing
Isolated Testing:
With composition, individual components can be tested independently of the␣
    ↪classes that use them. This isolation can simplify unit testing and increase␣
    ↪the reliability of tests, as you can focus on one piece of functionality at␣
    ↪a time.
5. Avoiding Inheritance Pitfalls
Fragile Base Class Problem:

Inheritance can lead to the fragile base class problem, where changes in a base␣
    ↪class inadvertently affect all derived classes. Composition mitigates this␣
    ↪risk by keeping components independent and reducing the interdependencies␣
    ↪that can complicate maintenance.
```

```
Multiple Inheritance Complexity:

Python supports multiple inheritance, but it can lead to complex scenarios and
↪ambiguity (e.g., the Diamond Problem). Composition avoids these issues by
↪providing a more straightforward and understandable way to build complex
↪behaviors without the complications of multiple inheritance.
```

[41]:
```python
# 5. How can you implement composition in Python classes? Provide examples of
↪using composition to create complex objects.
class CPU:
    def __init__(self, brand: str, speed: float):
        self.brand = brand
        self.speed = speed

    def get_info(self):
        return f"CPU: {self.brand}, Speed: {self.speed} GHz"


class RAM:
    def __init__(self, size: int):
        self.size = size

    def get_info(self):
        return f"RAM: {self.size} GB"


class Storage:
    def __init__(self, capacity: int):
        self.capacity = capacity

    def get_info(self):
        return f"Storage: {self.capacity} GB"


class Computer:
    def __init__(self, cpu: CPU, ram: RAM, storage: Storage):
        self.cpu = cpu
        self.ram = ram
        self.storage = storage

    def get_specs(self):
        return f"{self.cpu.get_info()}, {self.ram.get_info()}, {self.storage.
↪get_info()}"


if __name__ == "__main__":
    cpu = CPU("Intel", 3.5)
```

```
    ram = RAM(16)
    storage = Storage(512)

    computer = Computer(cpu, ram, storage)
    print(computer.get_specs())
```

```
CPU: Intel, Speed: 3.5 GHz, RAM: 16 GB, Storage: 512 GB
```

[43]:
```
#6.  Create a Python class hierarchy for a music player system, using␣
 ↪composition to represent playlists and songs.
class Song:
    def __init__(self, title: str, artist: str, duration: float):
        self.title = title
        self.artist = artist
        self.duration = duration

    def __str__(self):
        return f"'{self.title}' by {self.artist} ({self.duration} min)"


class Playlist:
    def __init__(self, name: str):
        self.name = name
        self.songs = []

    def add_song(self, song: Song):
        self.songs.append(song)

    def remove_song(self, song: Song):
        self.songs.remove(song)

    def play(self):
        print(f"Playing playlist: {self.name}")
        for song in self.songs:
            print(f"  - {song}")

    def total_duration(self):
        return sum(song.duration for song in self.songs)

    def __str__(self):
        return f"Playlist: {self.name}, Total Duration: {self.total_duration()}␣
 ↪min"


class MusicPlayer:
    def __init__(self):
        self.playlists = []
```

```
    def add_playlist(self, playlist: Playlist):
        self.playlists.append(playlist)

    def play_playlist(self, playlist: Playlist):
        playlist.play()

    def show_playlists(self):
        print("Playlists:")
        for playlist in self.playlists:
            print(f" - {playlist}")

if __name__ == "__main__":
    song1 = Song("Song One", "Artist A", 3.5)
    song2 = Song("Song Two", "Artist B", 4.2)
    song3 = Song("Song Three", "Artist C", 2.8)

    playlist1 = Playlist("My Favorite Songs")
    playlist1.add_song(song1)
    playlist1.add_song(song2)

    playlist2 = Playlist("Chill Vibes")
    playlist2.add_song(song3)

    music_player = MusicPlayer()
    music_player.add_playlist(playlist1)
    music_player.add_playlist(playlist2)

    music_player.show_playlists()
    music_player.play_playlist(playlist1)
```

```
Playlists:
 - Playlist: My Favorite Songs, Total Duration: 7.7 min
 - Playlist: Chill Vibes, Total Duration: 2.8 min
Playing playlist: My Favorite Songs
  - 'Song One' by Artist A (3.5 min)
  - 'Song Two' by Artist B (4.2 min)
```

[45]:
```
# 7.  Explain the concept of "has-a" relationships in composition and how it␣
↪helps design software systems.
class Engine:
    def start(self):
        return "Engine starting..."

class Car:
    def __init__(self):
        self.engine = Engine()
```

```python
    def start(self):
        return self.engine.start() + " Car is ready to go!"

if __name__ == "__main__":
    my_car = Car()
    print(my_car.start())
```

Engine starting… Car is ready to go!

[47]:
```python
# 8. Create a Python class for a computer system, using composition to␣
↪represent components like CPU, RAM, and storage devices.
class CPU:
    def __init__(self, brand: str, cores: int, speed: float):
        self.brand = brand
        self.cores = cores
        self.speed = speed

    def __str__(self):
        return f"{self.brand} CPU with {self.cores} cores at {self.speed} GHz"


class RAM:
    def __init__(self, size: int):
        self.size = size

    def __str__(self):
        return f"{self.size} GB RAM"


class Storage:
    def __init__(self, capacity: int, type: str):
        self.capacity = capacity
        self.type = type

    def __str__(self):
        return f"{self.capacity} GB {self.type} Storage"


class Computer:
    def __init__(self, cpu: CPU, ram: RAM, storage: Storage):
        self.cpu = cpu
        self.ram = ram
        self.storage = storage

    def __str__(self):
```

```
        return f"Computer Specifications:\n - {self.cpu}\n - {self.ram}\n -␣
↪{self.storage}"


if __name__ == "__main__":
    cpu = CPU("Intel", 8, 3.6)
    ram = RAM(16)
    storage = Storage(512, "SSD")

    computer = Computer(cpu, ram, storage)

    print(computer)
```

```
Computer Specifications:
 - Intel CPU with 8 cores at 3.6 GHz
 - 16 GB RAM
 - 512 GB SSD Storage
```

[49]:
```
# 9.  Describe the concept of "delegation" in composition and how it simplifies␣
↪the design of complex systems.
class PrinterDevice:
    def print_document(self, content: str):
        print(f"Printing: {content}")


class Printer:
    def __init__(self, printer_device: PrinterDevice):
        self.printer_device = printer_device  # Delegation

    def print(self, content: str):
        self.printer_device.print_document(content)


if __name__ == "__main__":
    printer_device = PrinterDevice()

    printer = Printer(printer_device)
    printer.print("Hello, World!")
```

```
Printing: Hello, World!
```

[51]:
```
# 10.  Create a Python class for a car, using composition to represent␣
↪components like the engine, wheels, an transmission.
class Engine:
    def __init__(self, horsepower, type_of_engine):
        self.horsepower = horsepower
        self.type_of_engine = type_of_engine
```

```python
    def start(self):
        return f"{self.type_of_engine} engine with {self.horsepower} HP started.
↪"

    def stop(self):
        return f"{self.type_of_engine} engine stopped."


class Wheel:
    def __init__(self, size, type_of_wheel):
        self.size = size
        self.type_of_wheel = type_of_wheel

    def rotate(self):
        return f"{self.type_of_wheel} wheel of size {self.size} inches is␣
↪rotating."


class Transmission:
    def __init__(self, transmission_type):
        self.transmission_type = transmission_type

    def shift(self):
        return f"Transmission shifted to {self.transmission_type}."


class Car:
    def __init__(self, engine, wheels, transmission):
        self.engine = engine
        self.wheels = wheels
        self.transmission = transmission

    def start(self):
        return self.engine.start()

    def drive(self):
        wheel_rotations = [wheel.rotate() for wheel in self.wheels]
        return "\n".join(wheel_rotations) + "\n" + self.transmission.shift()

    def stop(self):
        return self.engine.stop()


if __name__ == "__main__":
    engine = Engine(300, "V8")
    wheels = [Wheel(18, "Alloy") for _ in range(4)]
    transmission = Transmission("Automatic")
```

```
    my_car = Car(engine, wheels, transmission)

    print(my_car.start())
    print(my_car.drive())
    print(my_car.stop())
```

```
V8 engine with 300 HP started.
Alloy wheel of size 18 inches is rotating.
Alloy wheel of size 18 inches is rotating.
Alloy wheel of size 18 inches is rotating.
Alloy wheel of size 18 inches is rotating.
Transmission shifted to Automatic.
V8 engine stopped.
```

[53]:
```
# 11. How can you encapsulate and hide the details of composed objects in␣
 ↪Python classes to maintain abstraction?
class Engine:
    def __init__(self, horsepower, type_of_engine):
        self.__horsepower = horsepower
        self.__type_of_engine = type_of_engine

    def start(self):
        return f"{self.__type_of_engine} engine with {self.__horsepower} HP␣
 ↪started."

    def stop(self):
        return f"{self.__type_of_engine} engine stopped."


class Wheel:
    def __init__(self, size, type_of_wheel):
        self.__size = size
        self.__type_of_wheel = type_of_wheel

    def rotate(self):
        return f"{self.__type_of_wheel} wheel of size {self.__size} inches is␣
 ↪rotating."


class Transmission:
    def __init__(self, transmission_type):
        self.__transmission_type = transmission_type

    def shift(self):
        return f"Transmission shifted to {self.__transmission_type}."
```

```python
class Car:
    def __init__(self, engine, wheels, transmission):
        self.__engine = engine
        self.__wheels = wheels
        self.__transmission = transmission

    def start(self):
        return self.__engine.start()

    def drive(self):
        wheel_rotations = [wheel.rotate() for wheel in self.__wheels]
        return "\n".join(wheel_rotations) + "\n" + self.__transmission.shift()

    def stop(self):
        return self.__engine.stop()

    @property
    def engine_type(self):
        return self.__engine._Engine__type_of_engine

    @property
    def wheel_count(self):
        return len(self.__wheels)


if __name__ == "__main__":
    engine = Engine(300, "V8")
    wheels = [Wheel(18, "Alloy") for _ in range(4)]
    transmission = Transmission("Automatic")

    my_car = Car(engine, wheels, transmission)

    print(my_car.start())
    print(my_car.drive())
    print(my_car.stop())
    print(f"Engine type: {my_car.engine_type}")
    print(f"Number of wheels: {my_car.wheel_count}")
```

```
V8 engine with 300 HP started.
Alloy wheel of size 18 inches is rotating.
Alloy wheel of size 18 inches is rotating.
Alloy wheel of size 18 inches is rotating.
Alloy wheel of size 18 inches is rotating.
Transmission shifted to Automatic.
V8 engine stopped.
Engine type: V8
Number of wheels: 4
```

[55]:
```python
# 12.  Create a Python class for a university course, using composition to␣
↪represent students, instructors, and course materials.
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id

    def get_info(self):
        return f"Student: {self.name}, ID: {self.student_id}"


class Instructor:
    def __init__(self, name, employee_id):
        self.name = name
        self.employee_id = employee_id

    def get_info(self):
        return f"Instructor: {self.name}, ID: {self.employee_id}"


class CourseMaterial:
    def __init__(self, title, material_type):
        self.title = title
        self.material_type = material_type

    def get_info(self):
        return f"Material: {self.title}, Type: {self.material_type}"


class UniversityCourse:
    def __init__(self, course_name, instructor, materials):
        self.course_name = course_name
        self.instructor = instructor
        self.students = []
        self.materials = materials

    def add_student(self, student):
        self.students.append(student)

    def get_course_info(self):
        instructor_info = self.instructor.get_info()
        materials_info = "\n".join(material.get_info() for material in self.
↪materials)
        students_info = "\n".join(student.get_info() for student in self.
↪students)

        return (
```

```python
            f"Course: {self.course_name}\n"
            f"{instructor_info}\n"
            f"Materials:\n{materials_info}\n"
            f"Students Enrolled:\n{students_info if students_info else 'No␣
↪students enrolled.'}"
        )


if __name__ == "__main__":
    instructor = Instructor("Dr. Smith", "EMP001")
    materials = [
        CourseMaterial("Introduction to Python", "Textbook"),
        CourseMaterial("Python Programming Exercises", "Workbook"),
    ]

    course = UniversityCourse("Python Programming 101", instructor, materials)

    course.add_student(Student("Alice Johnson", "S12345"))
    course.add_student(Student("Bob Lee", "S67890"))

    print(course.get_course_info())
```

```
Course: Python Programming 101
Instructor: Dr. Smith, ID: EMP001
Materials:
Material: Introduction to Python, Type: Textbook
Material: Python Programming Exercises, Type: Workbook
Students Enrolled:
Student: Alice Johnson, ID: S12345
Student: Bob Lee, ID: S67890
```

[ ]:
```python
# 13.  Discuss the challenges and drawbacks of composition, such as increased␣
↪complexity and potential for tight coupling between objects.
1. Increased Complexity
Understanding Relationships: As you compose more objects, understanding the␣
↪relationships between them can become complex. Developers need to grasp not␣
↪just individual components but how they interact, which can lead to a␣
↪steeper learning curve.
More Interfaces: Each component often requires its own interface, leading to a␣
↪proliferation of interfaces that can be cumbersome to manage and document.
2. Potential for Tight Coupling
Interdependencies: If not carefully managed, composed objects can become␣
↪tightly coupled. Changes in one component might necessitate changes in␣
↪others, reducing flexibility and making maintenance harder.
Shared State Issues: When components share state, it can lead to unpredictable␣
↪behavior, especially in multi-threaded environments. This can complicate␣
↪debugging and testing.
```

```
3. Overhead
Performance: The indirection introduced by composition can lead to performance
 ↪overhead. Each method call through a composed interface may incur additional
 ↪time compared to direct calls in a tightly integrated system.
Memory Usage: Composing many small objects can lead to higher memory usage
 ↪compared to a monolithic approach, especially if many objects are
 ↪short-lived.
4. Difficulty in Testing
Mocking Dependencies: When testing composed objects, creating mocks or stubs
 ↪for dependencies can be complex, especially if those dependencies have
 ↪intricate interrelationships.
Integration Testing: Testing the integration of various components can be
 ↪challenging, requiring comprehensive test cases to cover all interaction
 ↪scenarios.
5. Versioning Challenges
Component Updates: Updating a single component in a composition can lead to
 ↪compatibility issues with other components, requiring careful versioning and
 ↪management.
Backward Compatibility: Ensuring backward compatibility can be difficult,
 ↪particularly if changes in one component affect the behavior expected by
 ↪others.
6. Design Rigor
Over-Engineering: There's a risk of over-engineering solutions with
 ↪composition, leading to unnecessary complexity and effort for simple
 ↪problems.
Inflexibility in Simple Scenarios: In scenarios where a simpler inheritance
 ↪model could suffice, using composition might introduce unnecessary overhead.
```

[57]:
```python
# 14.  Create a Python class hierarchy for a restaurant system, using
 ↪composition to represent menus, dishes, and ingredients.
class Ingredient:
    def __init__(self, name, quantity, unit):
        self.name = name
        self.quantity = quantity
        self.unit = unit

    def __str__(self):
        return f"{self.quantity} {self.unit} of {self.name}"


class Dish:
    def __init__(self, name, ingredients):
        self.name = name
        self.ingredients = ingredients  # List of Ingredient objects

    def get_ingredients(self):
```

```python
        return ', '.join(str(ingredient) for ingredient in self.ingredients)

    def __str__(self):
        return f"{self.name}: {self.get_ingredients()}"


class Menu:
    def __init__(self, title):
        self.title = title
        self.dishes = []  # List of Dish objects

    def add_dish(self, dish):
        self.dishes.append(dish)

    def get_dishes(self):
        return '\n'.join(str(dish) for dish in self.dishes)

    def __str__(self):
        return f"Menu: {self.title}\n" + self.get_dishes()


class Restaurant:
    def __init__(self, name):
        self.name = name
        self.menus = []  # List of Menu objects

    def add_menu(self, menu):
        self.menus.append(menu)

    def get_menus(self):
        return '\n\n'.join(str(menu) for menu in self.menus)

    def __str__(self):
        return f"Restaurant: {self.name}\n" + self.get_menus()


if __name__ == "__main__":
    ingredient1 = Ingredient("Tomato", 2, "pieces")
    ingredient2 = Ingredient("Mozzarella", 150, "grams")
    ingredient3 = Ingredient("Basil", 10, "grams")

    margherita_pizza = Dish("Margherita Pizza", [ingredient1, ingredient2,
 ↪ingredient3])

    lunch_menu = Menu("Lunch Menu")
    lunch_menu.add_dish(margherita_pizza)
```

```
    my_restaurant = Restaurant("The Italian Bistro")
    my_restaurant.add_menu(lunch_menu)

    print(my_restaurant)
```

```
Restaurant: The Italian Bistro
Menu: Lunch Menu
Margherita Pizza: 2 pieces of Tomato, 150 grams of Mozzarella, 10 grams of Basil
```

[ ]:
```
# 15. . Explain how composition enhances code maintainability and modularity in␣
 ↪Python programs.
1. Encapsulation of Behavior
Separation of Concerns: By breaking down functionality into smaller,␣
 ↪self-contained components (like classes), each piece focuses on a specific␣
 ↪responsibility. This separation makes it easier to understand, test, and␣
 ↪modify individual components without affecting others.
2. Reusability
Flexible Building Blocks: Components created through composition can be reused␣
 ↪in different contexts. For example, a class representing a logging mechanism␣
 ↪can be composed into various applications, reducing code duplication and␣
 ↪enhancing maintainability.
3. Easier Testing
Isolated Components: Since composed components can be tested independently,␣
 ↪unit testing becomes more straightforward. You can test each class in␣
 ↪isolation, leading to quicker identification of issues and making the␣
 ↪overall testing process more efficient.
4. Reduced Complexity
Manageable Codebases: Large codebases can become unwieldy if everything is␣
 ↪tightly coupled. Composition allows developers to create smaller, manageable␣
 ↪pieces of code that can be understood and modified independently, leading to␣
 ↪simpler overall architecture.
5. Improved Flexibility
Dynamic Composition: Composition allows for dynamic behavior changes at runtime.
 ↪ For instance, you can swap out components (like a logging strategy or a␣
 ↪data source) without altering the overall system structure, enabling quicker␣
 ↪adaptations to changing requirements.
6. Decoupling
Lower Coupling: Components can be designed to depend on interfaces rather than␣
 ↪concrete implementations. This decoupling means changes to one component␣
 ↪don’t necessitate widespread changes across the codebase, enhancing␣
 ↪maintainability and adaptability.
7. Facilitating Refactoring
Simpler Refactorings: When refactoring is needed, composed components can often␣
 ↪be modified or replaced independently of one another. This capability makes␣
 ↪the process less risky and more manageable.
8. Clearer Design
```

```
Intentional Structure: Composition promotes clearer design patterns, such as␣
↪the use of aggregates or service classes. This intentional structure helps␣
↪new developers understand the codebase more quickly and reduces onboarding␣
↪time.
9. Adaptability to Change
Easy Extensions: New functionality can often be added by creating new␣
↪components or extending existing ones, rather than altering the existing␣
↪structure. This adaptability helps keep the codebase responsive to new␣
↪requirements or changes in business logic.
```

[59]:

```
# 16. Create a Python class for a computer game character, using composition to␣
↪represent attributes like weapons, armor, and inventory.
class Weapon:
    def __init__(self, name, damage):
        self.name = name
        self.damage = damage

    def __str__(self):
        return f"{self.name} (Damage: {self.damage})"


class Armor:
    def __init__(self, name, defense):
        self.name = name
        self.defense = defense

    def __str__(self):
        return f"{self.name} (Defense: {self.defense})"


class Inventory:
    def __init__(self):
        self.items = []  # List to store items

    def add_item(self, item):
        self.items.append(item)

    def remove_item(self, item):
        self.items.remove(item)

    def get_items(self):
        return ', '.join(str(item) for item in self.items) if self.items else␣
↪"Empty"

    def __str__(self):
        return f"Inventory: {self.get_items()}"
```

```python
class Character:
    def __init__(self, name, health):
        self.name = name
        self.health = health
        self.weapon = None  # Initially no weapon
        self.armor = None  # Initially no armor
        self.inventory = Inventory()  # Composed inventory

    def equip_weapon(self, weapon):
        self.weapon = weapon

    def equip_armor(self, armor):
        self.armor = armor

    def take_damage(self, amount):
        if self.armor:
            amount -= self.armor.defense
            amount = max(amount, 0)  # Prevent negative damage
        self.health -= amount

    def __str__(self):
        weapon_info = str(self.weapon) if self.weapon else "No weapon equipped"
        armor_info = str(self.armor) if self.armor else "No armor equipped"
        return (f"Character: {self.name}\n"
                f"Health: {self.health}\n"
                f"Weapon: {weapon_info}\n"
                f"Armor: {armor_info}\n"
                f"{self.inventory}")


if __name__ == "__main__":
    sword = Weapon("Sword", 15)
    shield = Armor("Shield", 5)

    hero = Character("Knight", 100)

    hero.equip_weapon(sword)
    hero.equip_armor(shield)

    hero.inventory.add_item("Health Potion")
    hero.inventory.add_item("Mana Potion")

    print(hero)

    hero.take_damage(20)
    print("\nAfter taking damage:")
```

```python
    print(hero)
```

```
Character: Knight
Health: 100
Weapon: Sword (Damage: 15)
Armor: Shield (Defense: 5)
Inventory: Health Potion, Mana Potion

After taking damage:
Character: Knight
Health: 85
Weapon: Sword (Damage: 15)
Armor: Shield (Defense: 5)
Inventory: Health Potion, Mana Potion
```

[61]:
```python
# 17.Describe the concept of "aggregation" in composition and how it differs
  ↪from simple composition.
Composition vs. Aggregation
Ownership and Lifecycle

Composition: In composition, the contained objects (components) are strongly
  ↪dependent on the parent object for their lifecycle. If the parent object is
  ↪destroyed, the contained objects are also destroyed. This indicates a strong
  ↪ownership relationship. For example, a House class might contain Room
  ↪objects; if the house is deleted, the rooms are also deleted.
Aggregation: In aggregation, the contained objects can exist independently of
  ↪the parent object. The parent holds a reference to the contained objects but
  ↪does not have strict ownership over them. For example, a Library class might
  ↪contain Book objects; if the library is deleted, the books can still exist
  ↪independently, as they may be used elsewhere (like in a home).
Relationship Strength

Composition: Represents a strong relationship where the existence of the parts
  ↪is tied to the whole. It implies that the components are integral to the
  ↪whole and cannot meaningfully exist without it.
Aggregation: Represents a weaker relationship. The contained objects can exist
  ↪without the parent, which means they can be shared across multiple parents
  ↪or used in different contexts.
Use Cases

Composition: Used when the contained objects are specifically designed to be
  ↪part of the parent object and do not make sense outside of it. For example,
  ↪a Car class might have Engine and Wheel classes as components.
Aggregation: Used when the contained objects may belong to multiple parents or
  ↪have a shared lifecycle. For instance, a Teacher class might aggregate
  ↪Course objects, where each course can be taught by multiple teachers and
  ↪exist independently of any one teacher.
```

```python
class Engine:
    def __init__(self, type):
        self.type = type

class Car:
    def __init__(self):
        self.engine = Engine("V8")  # Strong ownership, Engine cannot exist␣
↪without Car

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

class Library:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

my_car = Car()
my_book = Book("1984")
my_library = Library()
my_library.add_book(my_book)
```

[63]:
```python
# 18. Create a Python class for a house, using composition to represent rooms,␣
↪furniture, and appliances.
class Furniture:
    def __init__(self, name, material):
        self.name = name
        self.material = material

    def __str__(self):
        return f"{self.name} (Material: {self.material})"


class Appliance:
    def __init__(self, name, power):
        self.name = name
        self.power = power
    def __str__(self):
        return f"{self.name} (Power: {self.power}W)"


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

```python
        self.furniture = []
        self.appliances = []

    def add_furniture(self, furniture):
        self.furniture.append(furniture)

    def add_appliance(self, appliance):
        self.appliances.append(appliance)

    def get_details(self):
        furniture_details = ', '.join(str(item) for item in self.furniture) if self.furniture else "No furniture"
        appliance_details = ', '.join(str(item) for item in self.appliances) if self.appliances else "No appliances"
        return (f"Room: {self.name}\n"
                f"Furniture: {furniture_details}\n"
                f"Appliances: {appliance_details}")


class House:
    def __init__(self, address):
        self.address = address
        self.rooms = []
    def add_room(self, room):
        self.rooms.append(room)

    def get_details(self):
        room_details = '\n\n'.join(room.get_details() for room in self.rooms) if self.rooms else "No rooms"
        return f"House Address: {self.address}\n\n{room_details}"


if __name__ == "__main__":
    sofa = Furniture("Sofa", "Leather")
    table = Furniture("Dining Table", "Wood")
    refrigerator = Appliance("Refrigerator", 150)
    microwave = Appliance("Microwave", 800)

    living_room = Room("Living Room")
    living_room.add_furniture(sofa)
    living_room.add_appliance(microwave)

    kitchen = Room("Kitchen")
    kitchen.add_furniture(table)
    kitchen.add_appliance(refrigerator)

    my_house = House("123 Main St")
```

```
    my_house.add_room(living_room)
    my_house.add_room(kitchen)

    print(my_house.get_details())
```

```
House Address: 123 Main St

Room: Living Room
Furniture: Sofa (Material: Leather)
Appliances: Microwave (Power: 800W)

Room: Kitchen
Furniture: Dining Table (Material: Wood)
Appliances: Refrigerator (Power: 150W)
```

[ ]:
```
# 19.  How can you achieve flexibility in composed objects by allowing them to␣
 ↪be replaced or modified dynamically at runtime?
Use of Interfaces and Abstract Base Classes
Defining Interfaces: Create abstract base classes or interfaces that define␣
 ↪expected behaviors. This allows different implementations to be swapped in␣
 ↪without changing the consuming code.
from abc import ABC, abstractmethod

class Appliance(ABC):
    @abstractmethod
    def operate(self):
        pass

class Refrigerator(Appliance):
    def operate(self):
        return "Cooling food"

class Microwave(Appliance):
    def operate(self):
        return "Heating food"


Composition Over Inheritance
Favor Composition: Instead of inheriting behavior, use composition to include␣
 ↪different behaviors. This allows you to change behaviors at runtime by␣
 ↪swapping composed objects.

class Kitchen:
    def __init__(self, appliance):
        self.appliance = appliance

    def use_appliance(self):
```

```
        return self.appliance.operate()

Dependency Injection
Inject Dependencies: Pass dependencies (like composed objects) to the␣
↪constructor or methods rather than hardcoding them. This way, you can␣
↪replace them easily without modifying the existing code.
my_kitchen = Kitchen(Refrigerator())
print(my_kitchen.use_appliance())  # "Cooling food"

my_kitchen.appliance = Microwave()  # Replacing the appliance
print(my_kitchen.use_appliance())  # "Heating food"


Configuration and Factory Patterns
Use Factory Methods: Implement factory patterns to create and configure objects.
↪ You can change configurations or types without modifying the client code.
class ApplianceFactory:
    @staticmethod
    def create_appliance(type):
        if type == "refrigerator":
            return Refrigerator()
        elif type == "microwave":
            return Microwave()

appliance = ApplianceFactory.create_appliance("microwave")
kitchen = Kitchen(appliance)

Dynamic Object Replacement
Allow for Runtime Replacement: Design your classes so that their internal state␣
↪can be changed at runtime. This could involve using setter methods or␣
↪properties to change composed objects.

class Kitchen:
    def __init__(self, appliance):
        self.appliance = appliance

    def change_appliance(self, new_appliance):
        self.appliance = new_appliance

# Usage
kitchen = Kitchen(Refrigerator())
print(kitchen.use_appliance())  # "Cooling food"

kitchen.change_appliance(Microwave())  # Dynamically replace appliance
print(kitchen.use_appliance())  # "Heating food"

Event-Driven Architecture
```

```
Use Events for Modifications: Implement an event-driven system where changes in␣
↪one part of the application can trigger updates or replacements in composed␣
↪objects.


class EventManager:
    def __init__(self):
        self.listeners = []

    def subscribe(self, listener):
        self.listeners.append(listener)

    def notify(self, event):
        for listener in self.listeners:
            listener(event)

# Example listener could change appliance on event
```

[67]:

```
# 20. Create a Python class for a social media application, using composition␣
↪to represent users, posts, and comments.
class Comment:
    def __init__(self, user, content):
        self.user = user  # User who made the comment
        self.content = content

    def __str__(self):
        return f"{self.user.username}: {self.content}"


class Post:
    def __init__(self, user, content):
        self.user = user  # User who created the post
        self.content = content
        self.comments = []  # List to store Comment objects

    def add_comment(self, comment):
        self.comments.append(comment)

    def get_comments(self):
        return '\n'.join(str(comment) for comment in self.comments) if self.
↪comments else "No comments"

    def __str__(self):
        return (f"Post by {self.user.username}: {self.content}\n"
                f"Comments:\n{self.get_comments()}")
```

```python
class User:
    def __init__(self, username):
        self.username = username
        self.posts = []  # List to store Post objects

    def create_post(self, content):
        new_post = Post(self, content)
        self.posts.append(new_post)
        return new_post

    def __str__(self):
        return f"User: {self.username}\nPosts:\n" + '\n'.join(str(post) for post in self.posts) if self.posts else "No posts"


class SocialMedia:
    def __init__(self):
        self.users = []  # List to store User objects

    def add_user(self, username):
        user = User(username)
        self.users.append(user)
        return user

    def get_users(self):
        return '\n'.join(user.username for user in self.users) if self.users else "No users"

    def __str__(self):
        return f"Social Media Users:\n{self.get_users()}"


# Example usage
if __name__ == "__main__":
    # Create a social media application
    social_media_app = SocialMedia()

    # Add users
    alice = social_media_app.add_user("Alice")
    bob = social_media_app.add_user("Bob")

    # Users create posts
    post1 = alice.create_post("Hello, world!")
    post2 = bob.create_post("It's a sunny day!")

    # Users add comments
    post1.add_comment(Comment(bob, "Nice to see your first post!"))
```

```python
    post2.add_comment(Comment(alice, "Enjoy your day!"))

    # Print user details
    print(alice)
    print("\n" + str(bob))
```

```
User: Alice
Posts:
Post by Alice: Hello, world!
Comments:
Bob: Nice to see your first post!

User: Bob
Posts:
Post by Bob: It's a sunny day!
Comments:
Alice: Enjoy your day!
```

[ ]:
```python
#
```