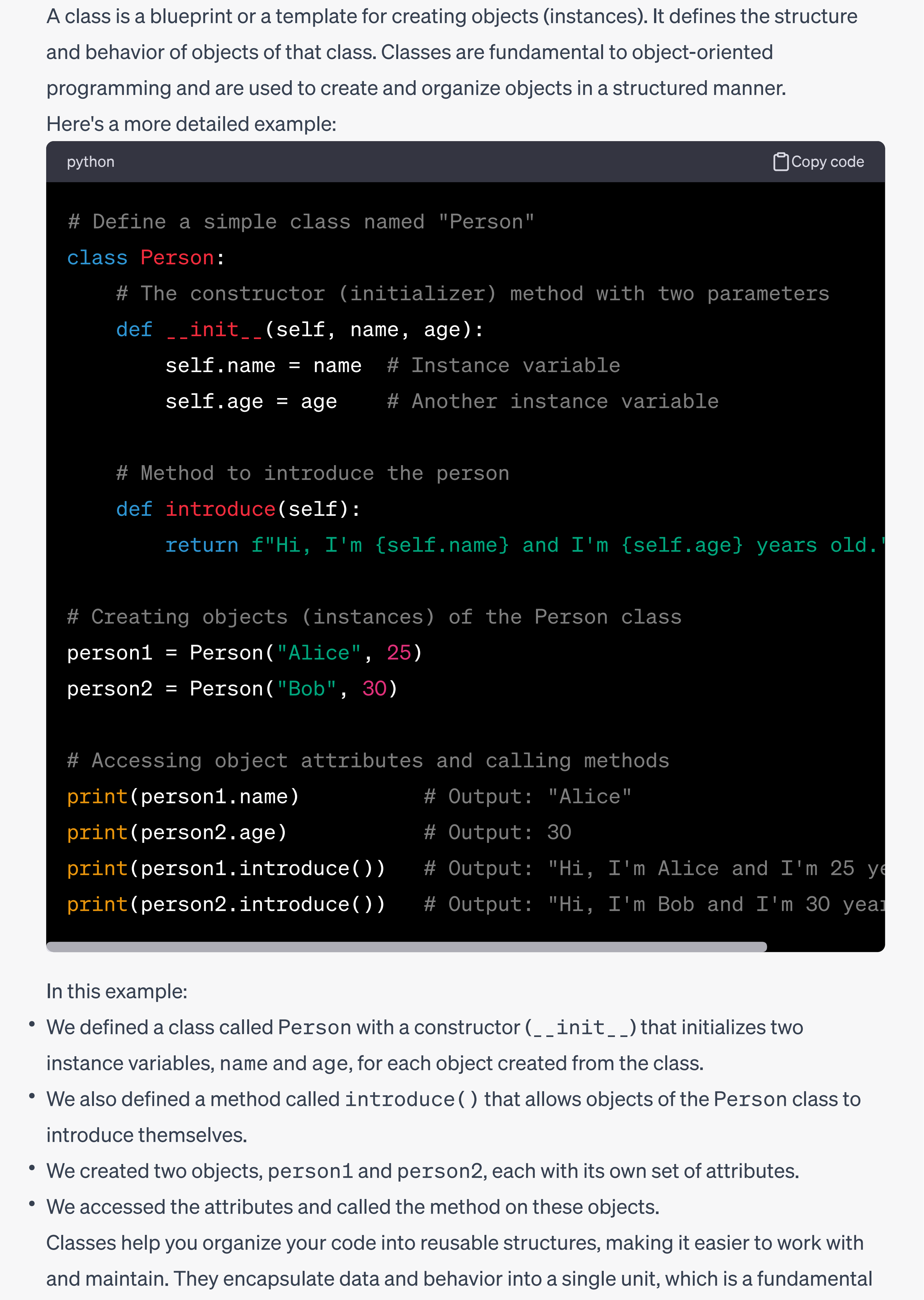


A class is a blueprint or a template for creating objects (instances). It defines the structure and behavior of objects of that class. Classes are fundamental to object-oriented programming and are used to create and organize objects in a structured manner.

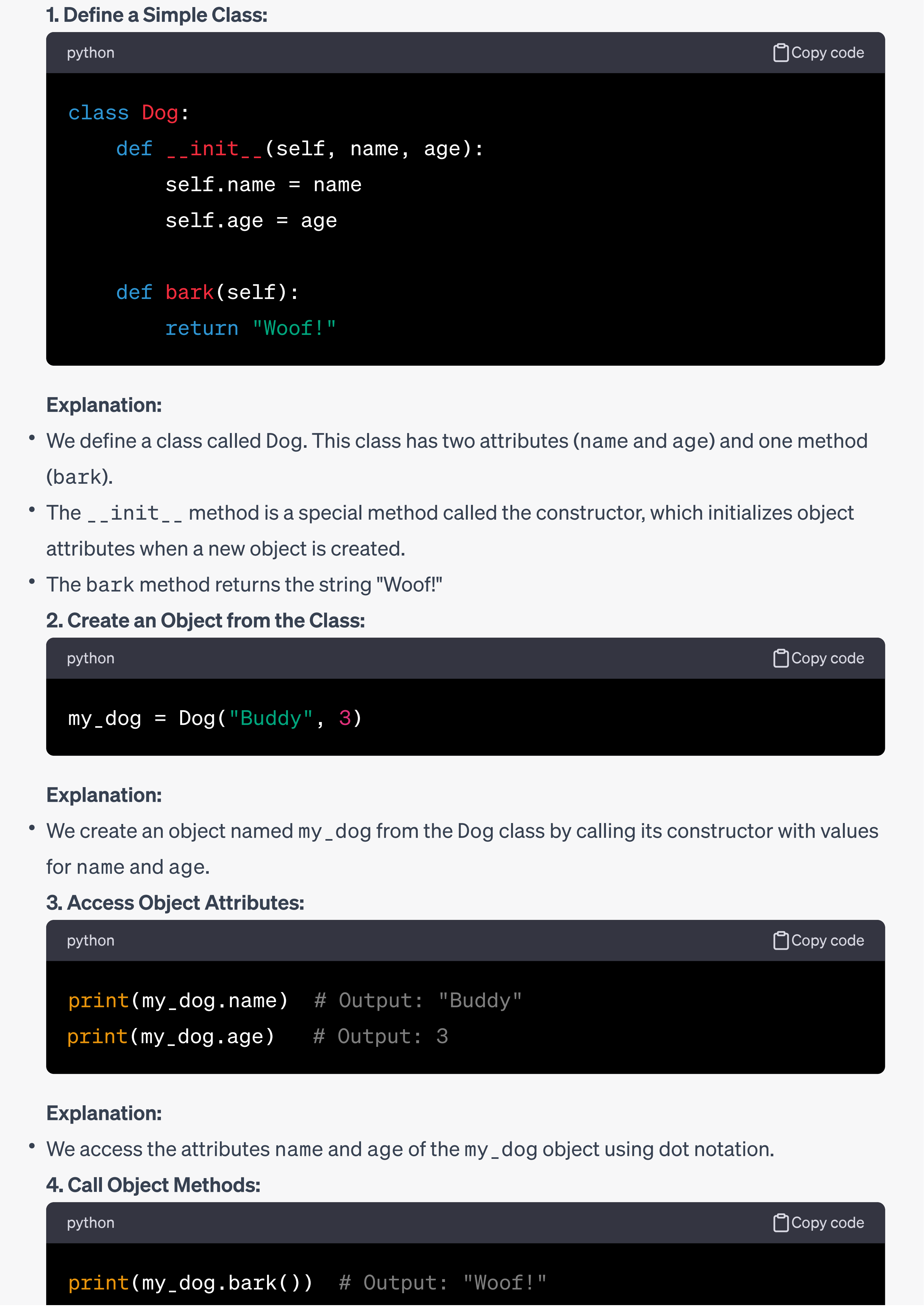
Here's a more detailed example:

```python
# Define a simple class named "Person"
class Person:
    # The constructor (initializer) method with two parameters
    def __init__(self, name, age):
        self.name = name  # Instance variable
        self.age = age    # Another instance variable

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

# Creating objects (instances) of the Person class
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

# Accessing object attributes and calling methods
print(person1.name)           # Output: "Alice"
print(person2.age)            # Output: 30
print(person1.introduce())    # Output: "Hi, I'm Alice and I'm 25 ye
print(person2.introduce())    # Output: "Hi, I'm Bob and I'm 30 yea
```

In this example:

- We defined a class called `Person` with a constructor (`__init__`) that initializes two instance variables, name and age, for each object created from the class.
- We also defined a method called `introduce()` that allows objects of the `Person` class to introduce themselves.
- We created two objects, `person1` and `person2`, each with its own set of attributes.
- We accessed the attributes and called the method on these objects.

Classes help you organize your code into reusable structures, making it easier to work with and maintain. They encapsulate data and behavior into a single unit, which is a fundamental



1. **Define a Simple Class:**

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

    def bark(self):
        return "Woof!"
```

Explanation:

- We define a class called Dog. This class has two attributes (`name` and `age`) and one method (`bark`).
- The `__init__` method is a special method called the constructor, which initializes object attributes when a new object is created.
- The `bark` method returns the string "Woof!"

2. **Create an Object from the Class:**

```python
my_dog = Dog("Buddy", 3)
```

Explanation:

- We create an object named `my_dog` from the Dog class by calling its constructor with values for name and age.

3. **Access Object Attributes:**

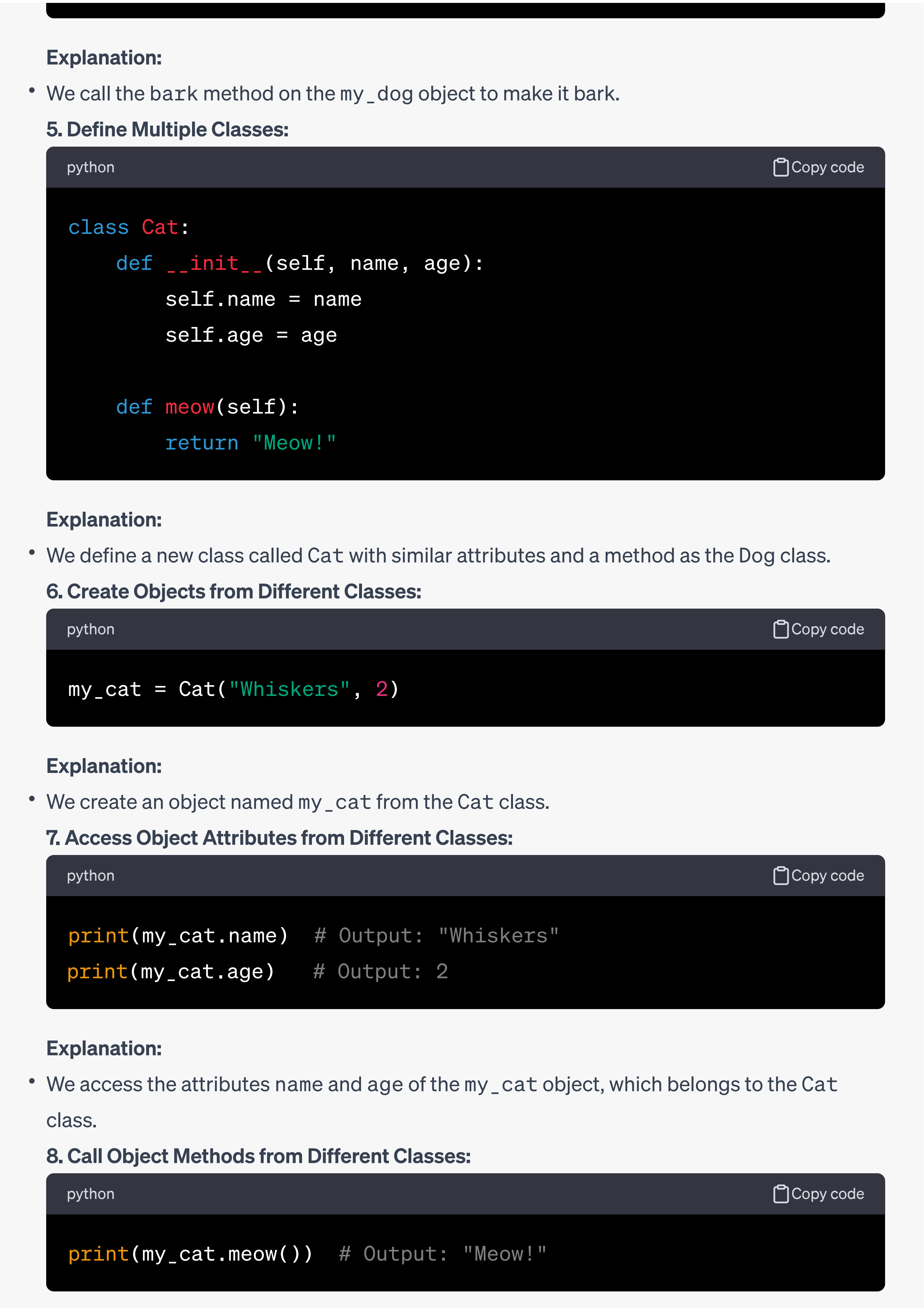
```python
print(my_dog.name)  # Output: "Buddy"
print(my_dog.age)   # Output: 3
```

Explanation:

- We access the attributes `name` and `age` of the `my_dog` object using dot notation.

4. **Call Object Methods:**

```python
print(my_dog.bark())  # Output: "Woof!"
```



Explanation:

- We call the bark method on the my_dog object to make it bark.

5. Define Multiple Classes:

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

    def meow(self):
        return "Meow!"
```

Explanation:

- We define a new class called Cat with similar attributes and a method as the Dog class.

6. Create Objects from Different Classes:

```python
my_cat = Cat("Whiskers", 2)
```

Explanation:

- We create an object named my_cat from the Cat class.

7. Access Object Attributes from Different Classes:

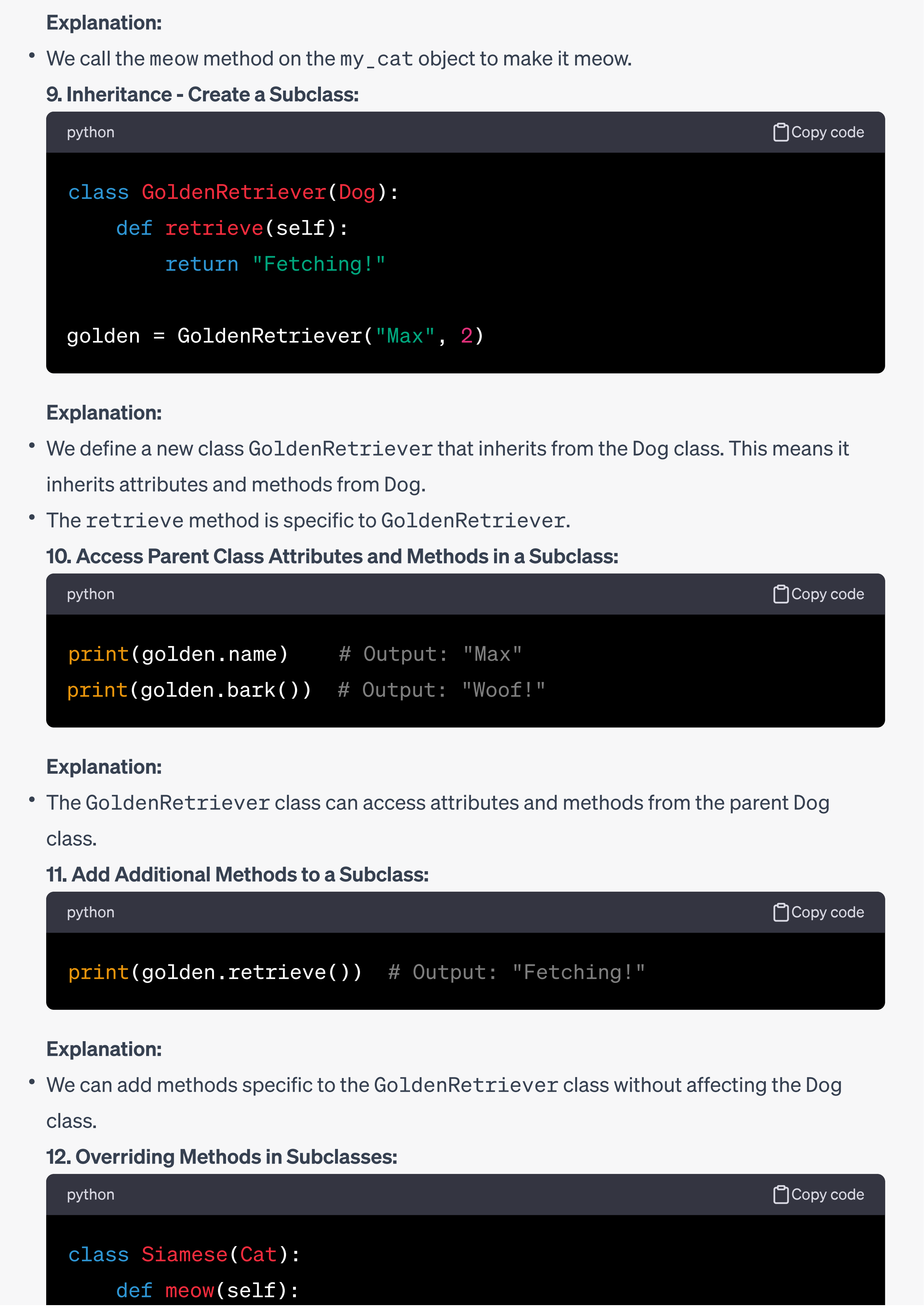
```python
print(my_cat.name)  # Output: "Whiskers"
print(my_cat.age)   # Output: 2
```

Explanation:

- We access the attributes name and age of the my_cat object, which belongs to the Cat class.

8. Call Object Methods from Different Classes:

```python
print(my_cat.meow())  # Output: "Meow!"
```



Explanation:

- We call the `meow` method on the `my_cat` object to make it meow.

9. Inheritance - Create a Subclass:

```python
class GoldenRetriever(Dog):
    def retrieve(self):
        return "Fetching!"

golden = GoldenRetriever("Max", 2)
```

Explanation:

- We define a new class `GoldenRetriever` that inherits from the Dog class. This means it inherits attributes and methods from Dog.
- The `retrieve` method is specific to `GoldenRetriever`.

10. Access Parent Class Attributes and Methods in a Subclass:

```python
print(golden.name)    # Output: "Max"
print(golden.bark())  # Output: "Woof!"
```

Explanation:

- The `GoldenRetriever` class can access attributes and methods from the parent Dog class.

11. Add Additional Methods to a Subclass:

```python
print(golden.retrieve())  # Output: "Fetching!"
```

Explanation:

- We can add methods specific to the `GoldenRetriever` class without affecting the Dog class.

12. Overriding Methods in Subclasses:

```python
class Siamese(Cat):
    def meow(self):
        return "Loud Meow!"

siamese_cat = Siamese("Mittens", 1)
print(siamese_cat.meow())  # Output: "Loud Meow!"
```

Explanation:

- The `Siamese` class overrides the `meow` method from the `Cat` class with its own implementation.

13. Initialize Objects with Default Values:

```python
class Car:
    def __init__(self, make="Unknown", model="Unknown"):
        self.make = make
        self.model = model

my_car = Car()
print(my_car.make)     # Output: "Unknown"
print(my_car.model)    # Output: "Unknown"
```

Explanation:

- We define default values for the `make` and `model` attributes in the constructor, allowing objects to be created without specifying these values.

14. Modify Object Attributes:

```python
my_car.make = "Toyota"
my_car.model = "Camry"
```

Explanation:

- We can change the values of object attributes after the object is created.

15. Mutable Objects as Attributes:

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

    def add_item(self, item):
        self.items.append(item)

cart = ShoppingCart()
cart.add_item("Book")
cart.add_item("Shoes")
print(cart.items)  # Output: ["Book", "Shoes"]
```

Explanation:

- The `ShoppingCart` class has a mutable list as an attribute to store items.

16. Immutable Objects as Attributes:

python Copy code

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

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

my_circle = Circle(5)
print(my_circle.area())  # Output: 78.5
```

Explanation:

- The `Circle` class has an immutable attribute `radius`.

17. Using Class Variables:

python Copy code

```
class Student:
    school_name = "ABC School"

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

```
student1 = Student("Alice", 9)
student2 = Student("Bob", 10)

print(student1.school_name)  # Output: "ABC School"
print(student2.school_name)  # Output: "ABC School"
```

Explanation:

- `school_name` is a class variable shared by all instances of the `Student` class.

18. Accessing Class Variables:

python Copy code

```
print(Student.school_name)  # Output: "ABC School"
```

Explanation:

- Class variables can be accessed using the class name.

19. Changing a Class Variable:

python Copy code

```
Student.school_name = "XYZ School"
print(student1.school_name)  # Output: "XYZ School"
print(student2.school_name)  # Output: "XYZ School"
```

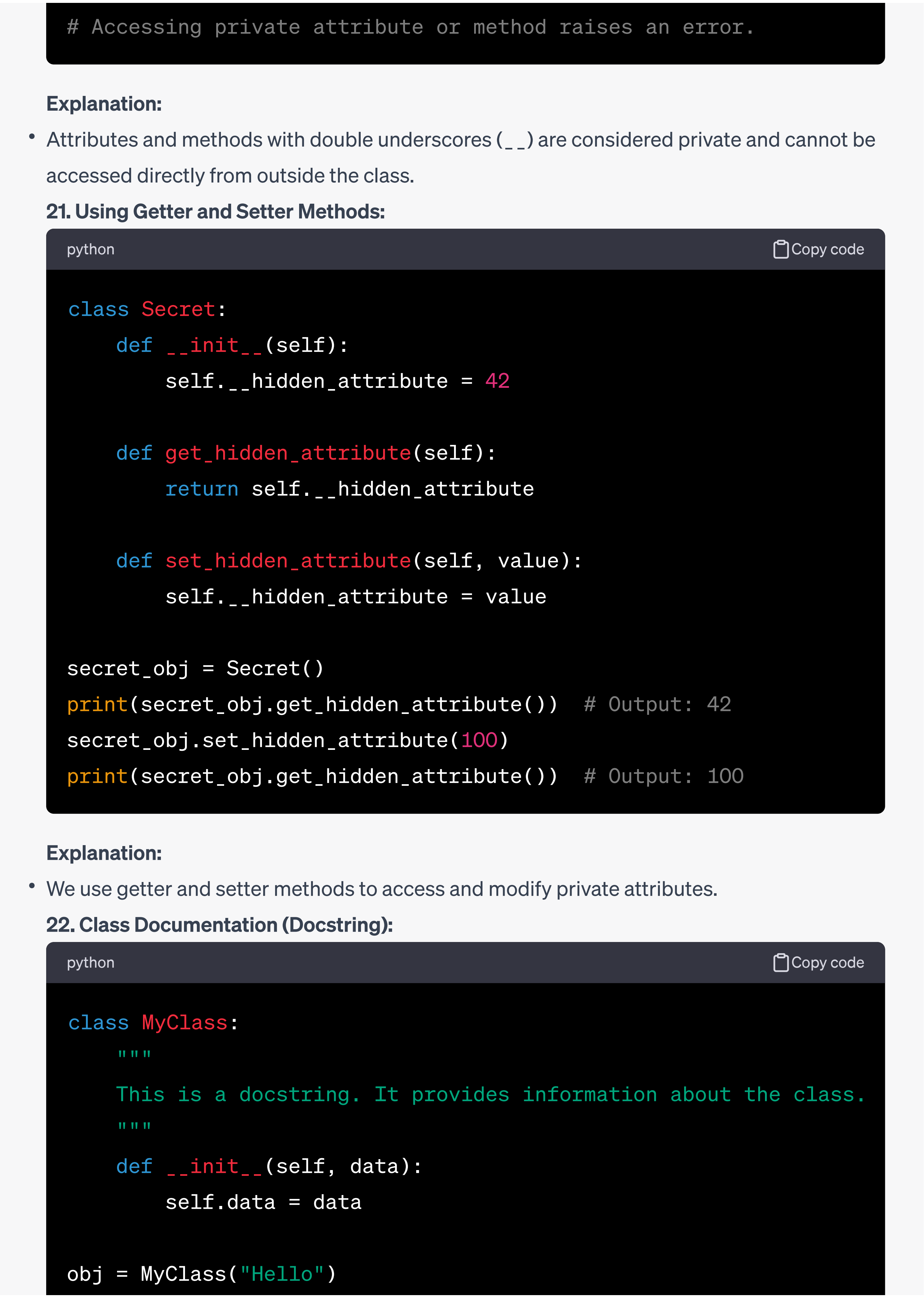
Explanation:

- Class variables can be changed for all instances of the class by modifying them using the class name.

20. Private Attributes and Methods:

python Copy code

```
class Secret:
    def __init__(self):
        self.__hidden_attribute = 42

    def __hidden_method(self):
        return "This is a secret method!"

secret_obj = Secret()
```



```
# Accessing private attribute or method raises an error.
```

Explanation:

- Attributes and methods with double underscores (__) are considered private and cannot be accessed directly from outside the class.

21. Using Getter and Setter Methods:

```python
class Secret:
    def __init__(self):
        self.__hidden_attribute = 42

    def get_hidden_attribute(self):
        return self.__hidden_attribute

    def set_hidden_attribute(self, value):
        self.__hidden_attribute = value

secret_obj = Secret()
print(secret_obj.get_hidden_attribute())  # Output: 42
secret_obj.set_hidden_attribute(100)
print(secret_obj.get_hidden_attribute())  # Output: 100
```

Explanation:

- We use getter and setter methods to access and modify private attributes.

22. Class Documentation (Docstring):

```python
class MyClass:
    """
    This is a docstring. It provides information about the class.
    """
    def __init__(self, data):
        self.data = data

obj = MyClass("Hello")
```

```
# Access docstring using help function: help(obj)
```

Explanation:

- Docstrings are used to provide documentation and information about the class. They can be accessed using the `help()` function.

These examples illustrate various aspects of classes in Python, from basic class creation to inheritance, encapsulation, and the use of class variables and documentation.