BOOLEAN

Object Oriented Programming in Python

Abul Umayer
Gonçalo Bernardo
Mahshid Pashootan
Marketa Chalupova
Morteza
Zeinab Toghani

BOOLEAN

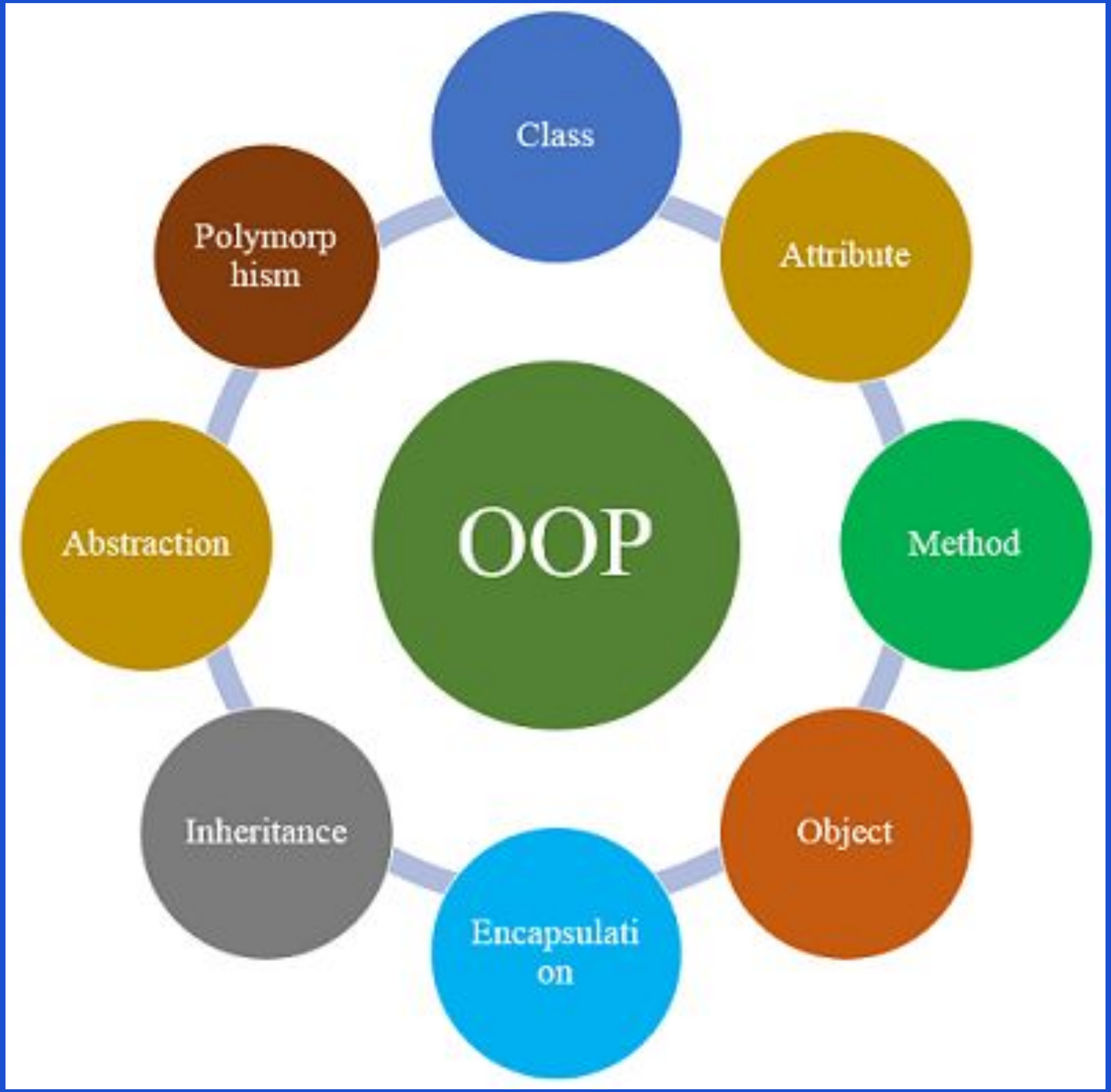
What We Will Talk About...

1-2	OOP and the four pillars	GB
3a-b	Class/ Instance - Attribute/method	MC
3c	Examples of Inheritance	M
3d-e	Examples of Encapsulation	MP
3f	Examples of abstraction	AU
3g	Overloading/ Overriding	ZT
4	Conclusions	-

BOOLEAN

1. What is OOP?
2. The four pillars

BOOLEAN



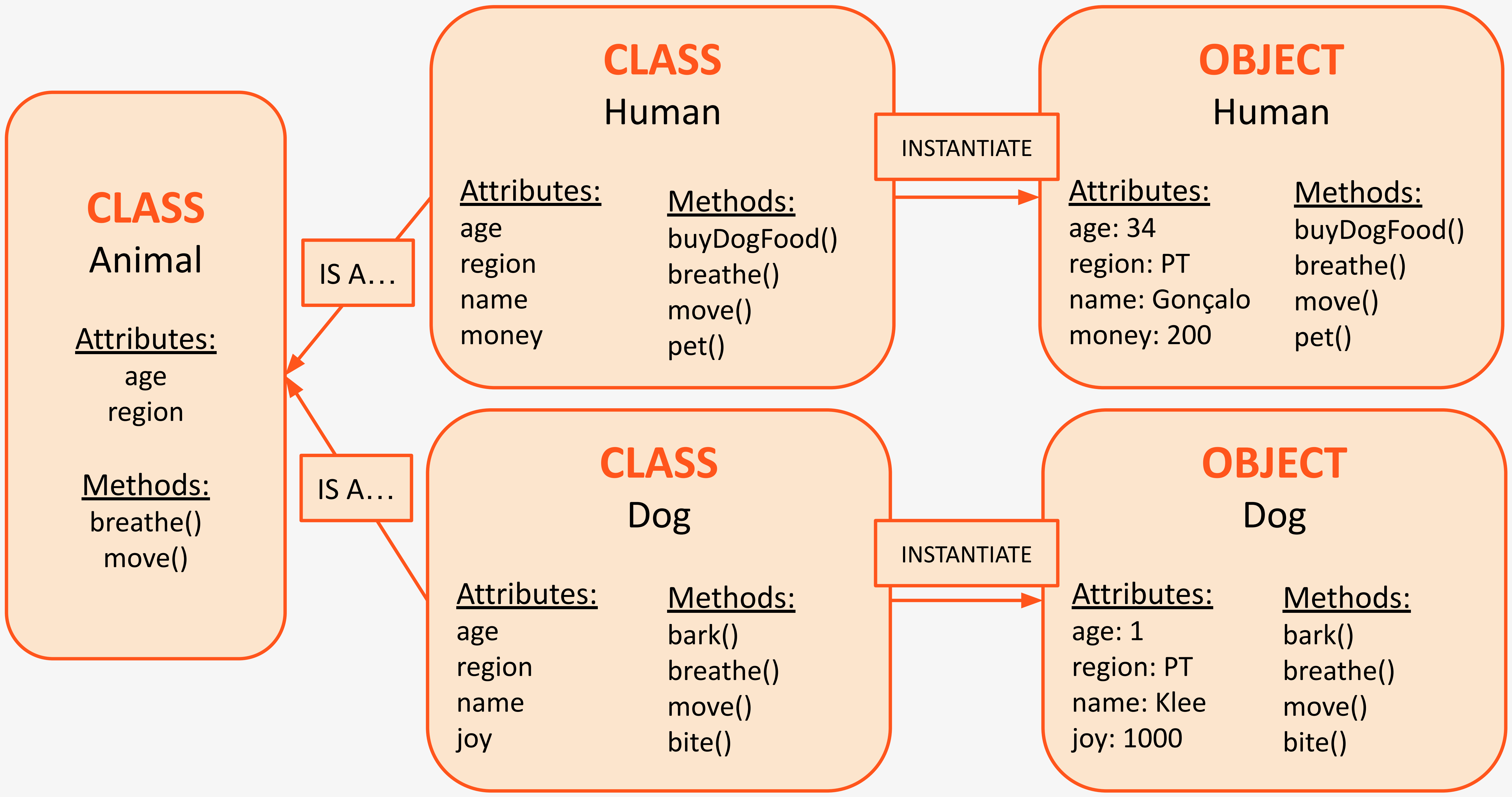
What is OOP?

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic (functional programming).

An object can be defined as a data field that has unique attributes and behavior

Simplifying with examples

OOP paradigm is like to simulate a way of programming based on entities that will interact with one another



BOOLEAN

Simplifying with examples

Now let's give an example how those two objects (Gonçalo and Klee) could interact!

OBJECT

Human (goncalo)

Attributes:	Methods:
age: 34	buyDogFood()
region: PT	breathe()
name: Gonçalo	move()
money: 200	pet()

OBJECT

Dog (klee)

Attributes:	Methods:
age: 1	bark()
region: PT	breathe()
name: Klee	move()
joy: 1000	bite()

EXAMPLE 1

(After being instantiated)

goncalo.breathe()
klee.breathe()

goncalo.pet()

klee.bark()

goncalo.buyDogFood()

goncalo.move() *
klee.move() *

pet() implementation:

pet:

goncalo.move()
goncalo.cuddle()

klee.joy=klee.joy+100

buyDogFood() implementation:

buyDogFood:

print: "He's buying food…"

goncalo.money=goncalo.money-50

*** .move()**

both objects can move, but they do it differently, so here we have a @override situation, ie, the implementation of this method normally with be defined on child classes Dog and Human and not in the Animal Class.

BOOLEAN

Simplifying with examples

Now we can talk about the 4 pillars of OOP!

OBJECT
Human (goncalo)

Attributes:	Methods:
age: 34	buyDogFood()
region: PT	breathe()
name: Gonçalo	move()
money: 200	pet()

OBJECT
Dog (klee)

Attributes:	Methods:
age: 1	bark()
region: PT	breathe()
name: Klee	move()
joy: 1000	bite()

INHERITANCE
In the example Human and Dog inherit from Animal Class, so they **always** must inherit parent’s attributes and methods. If the way each one of the objects do something is different from another, we override the implementation of the parent class. This pillar is useful for not repeating unnecessary code.

ENCAPSULATION
This example don’t cover encapsulation clearly, but OOP languages must have a mechanism to encapsulate attributes or methods, ie, make the access to them restrict.
For example, in this example I could encapsulate the attribute money from Human Class because it makes sense that I don’t want to make human’s money public. Another important thing that makes encapsulation useful is if I don’t want to make “public” the implementation of my methods.

ABSTRACTION
Similarly with encapsulation, abstraction permits the coders to hide implementations, This is important if we want to show only the signatures of methods to the “client”, who don’t need to know how the process works under the woods. Some languages make use of interfaces and abstract classes to implement abstraction.

POLYMORPHISM
As the name says: “various forms”. This so important pillar of OOP is strictly related with inheritance. In OOP languages is very useful to define the behaviour of the object in run time. For example, here I could create the objects goncalo and klee, but somewhere in the code I just refer to the object as an Animal and jut in run-time the program will define which implementation it will perform! We call this override.
Another type of polymorphism is when we create methods with the same name in the same class, but with different parameters. In that case we have overloading and it occurs in compilation time.

BOOLEAN

3.a Class instance
3.b Attribute/Method

BOOLEAN

Class and Instances

Class

= a template that defines the properties (attributes) and behaviors (methods) that its object will have

- reusable template - remove repetition
- new class creates a new type of object

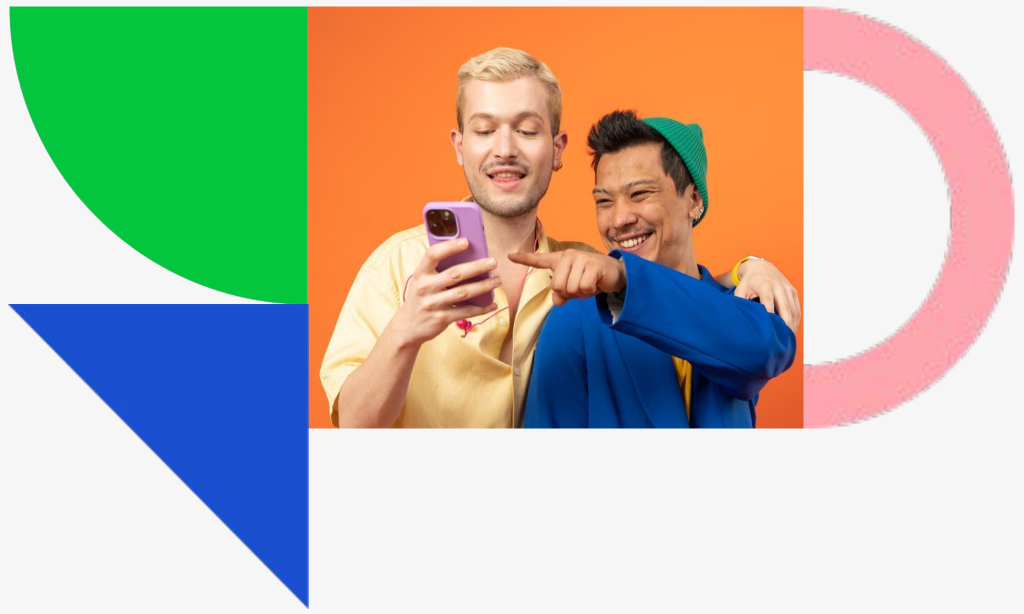
Instance

= a specific object created from a class

- has unique data
- can access methods and attributes defined in the class

BOOLEAN

Methods and Attributes



Methods

= functions defined within a class

- define behavior associated with the objects of the class
- **Instance methods** and **Class methods**

Instance methods - operate on instance-specific data

- most common type of methods
- defined within a class and take ‘**self**’ as their first parameters to access instance attributes and other methods

Class methods - operate on class-level data

- typically used for tasks that involve the entire class, not just specific instance

Attributes

= variables that store data related to a class or an instance of the class

- **Class attributes** and **Instance attributes**

Class attributes - shared among all instance of a class

- often used to store information that applies to the entire class

Instance attributes - specific to each instance of a class

- typically defined within the **class constructor** (‘**__init__**’ method)
- represent unique data for each object
- attribute shadowing

BOOLEAN

The '__init__' constructor

the '**__init__**' method is a special method that is automatically called when you create an instance of a class

- primary function = to **init**ialize the attributes of the class

- always takes at least one parameters, conventionally named **'self'**
- 'self' refers to the instance being created and is automatically passed by Python when you create an object from the class

```python
# Define a class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Create instances of the class and initialize attributes
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

# Access the attributes
print(person1.name)  # Output: Alice
print(person2.age)   # Output: 25
```

BOOLEAN

3.c Inheritance

BOOLEAN

Inheritance

- **It allows new classes to be based on existing classes, inheriting their attributes and methods**
- **The existing class is called the parent or superclass**
- **The new class is called the child or subclass**

Imagine a family scenario:

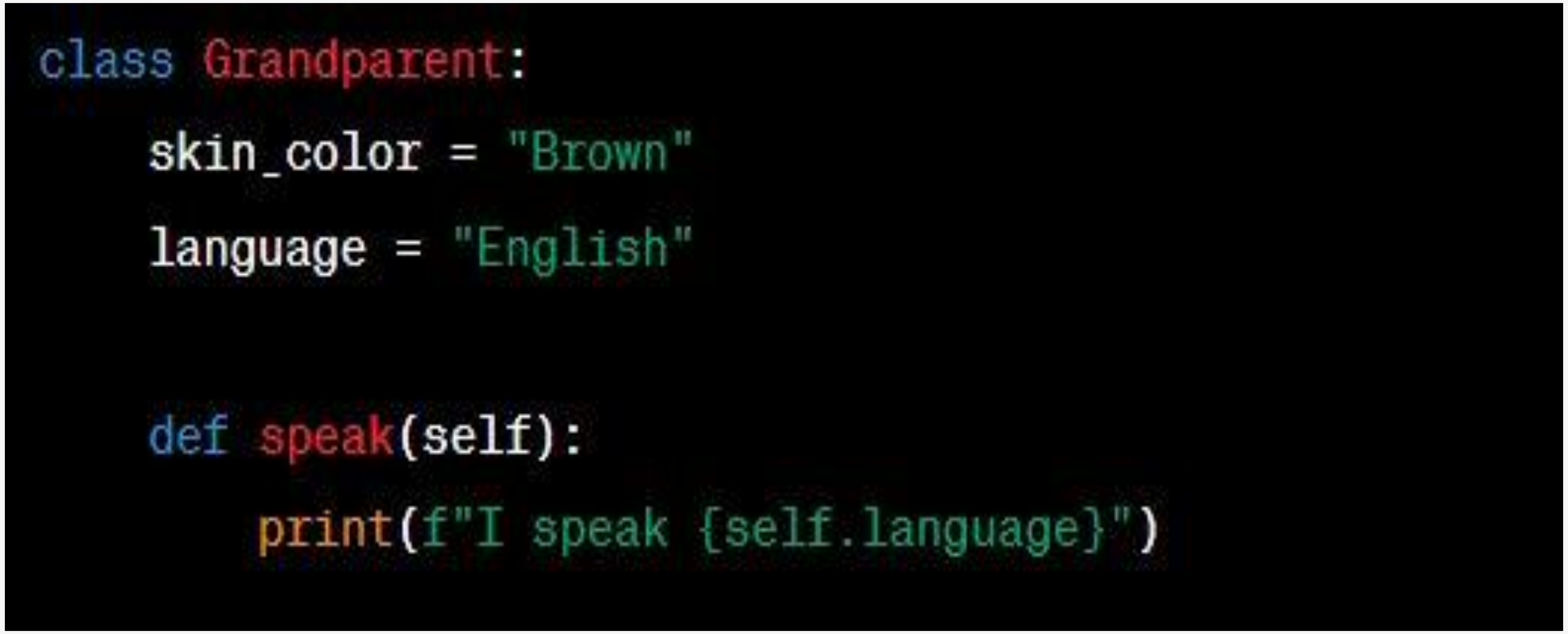
- **Grandparent: Think of the grandparent as having general traits like skin color, height, or even a specific language they speak**

- **Parent: The parent, being the child of the grandparent, will naturally inherit these traits. However, the parent might also have some additional traits or skills they've picked up, like playing a musical instrument.**

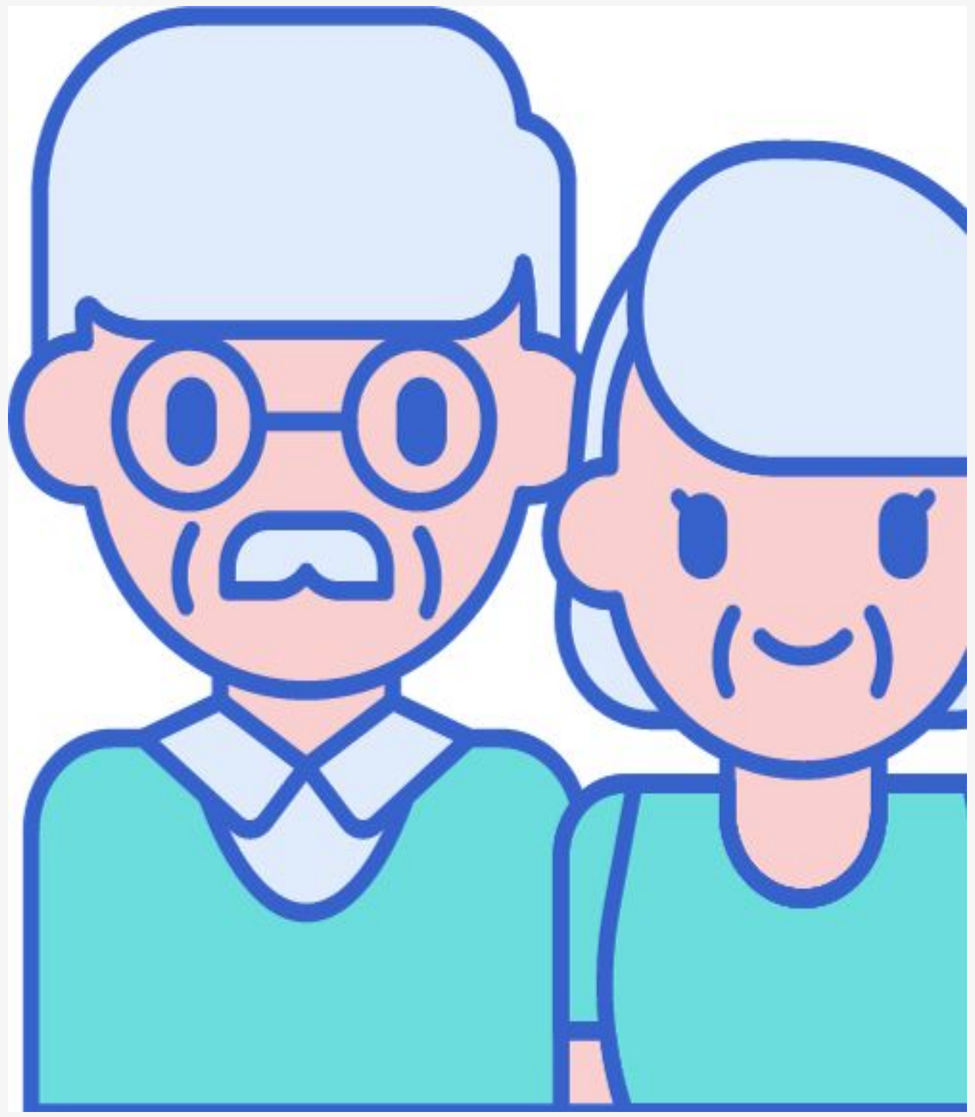
- **Child: The child, in turn, inherits traits from both the grandparent and parent. For instance, they might inherit the skin color from the grandparent, the musical talent from the parent, and yet they might have some unique traits of their own, like a passion for a particular sport.**

❖ BOOLEAN

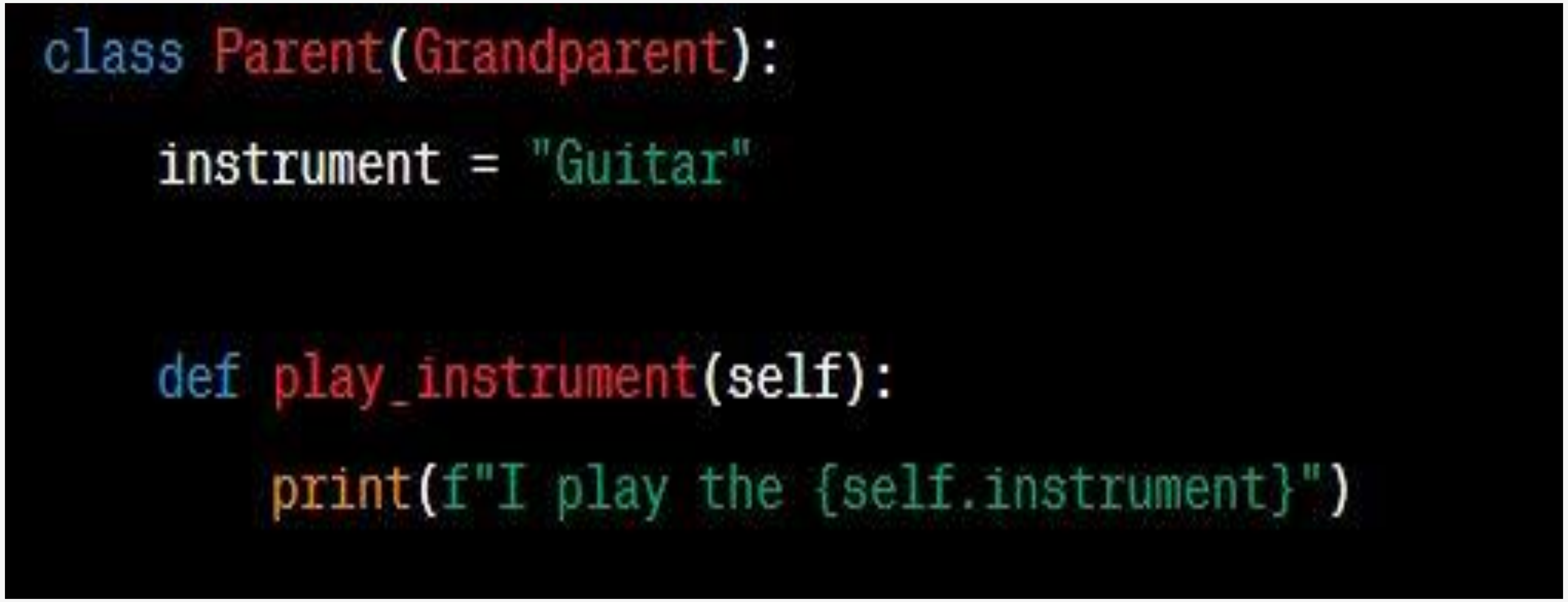
# This is our base class representing the grandparent



```
class Grandparent:
    skin_color = "Brown"
    language = "English"

    def speak(self):
        print(f"I speak {self.language}")
```



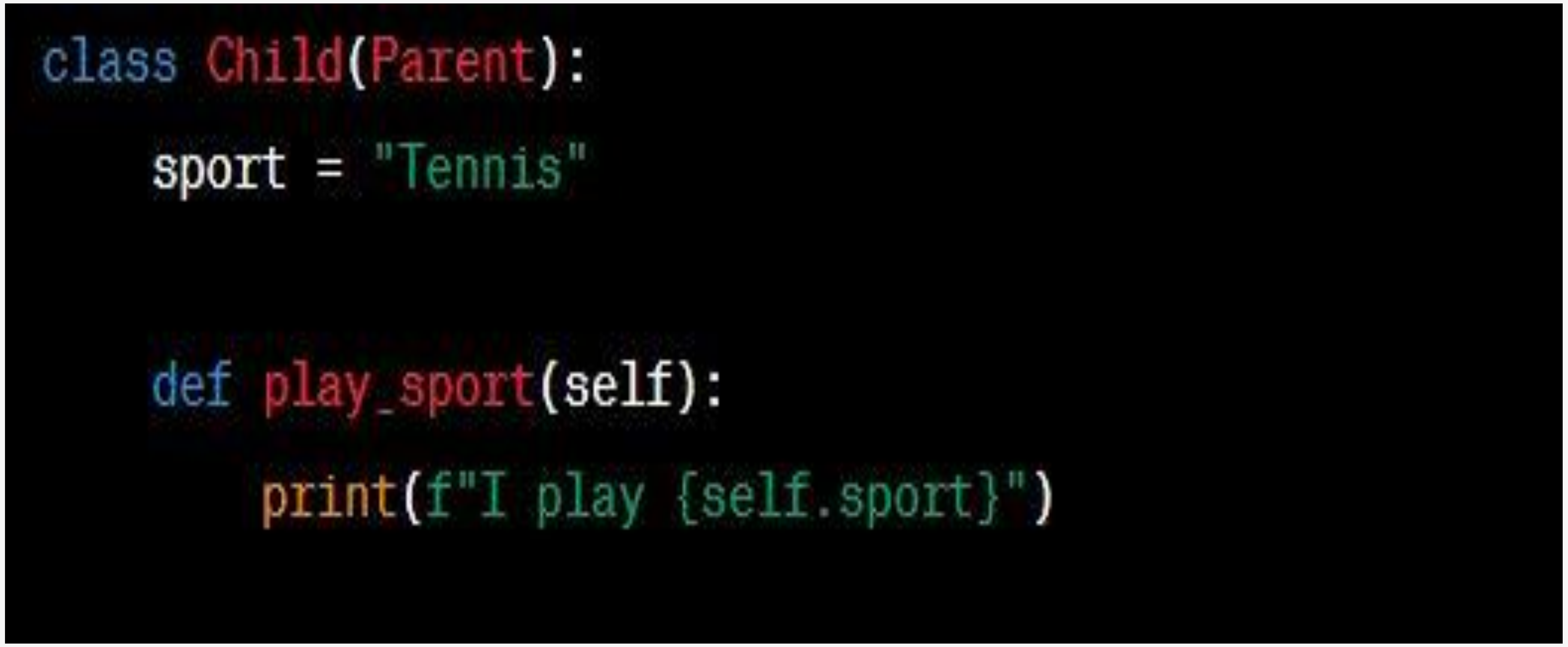
# Parent class inherits from Grandparent



```
class Parent(Grandparent):
    instrument = "Guitar"

    def play_instrument(self):
        print(f"I play the {self.instrument}")
```

❖ BOOLEAN

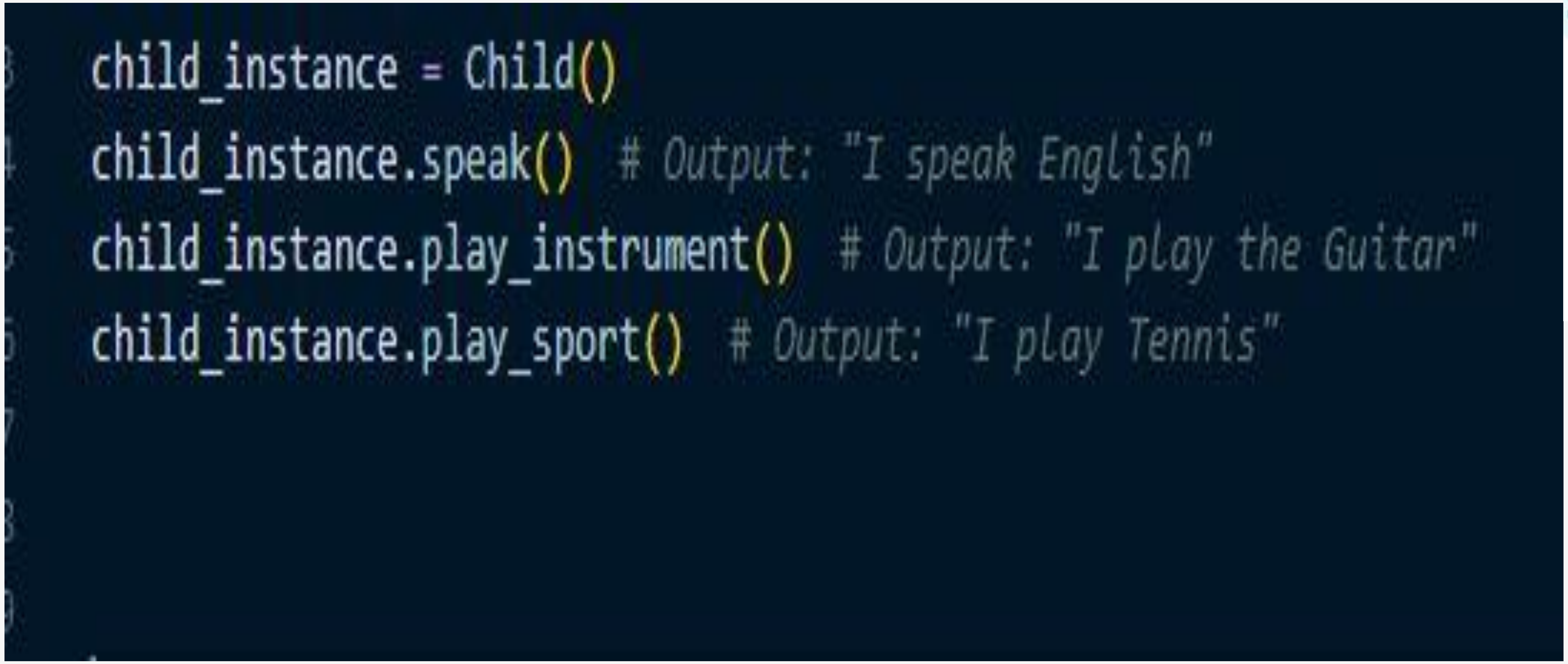
# Child class inherits from Parent (and indirectly from Grandparent)



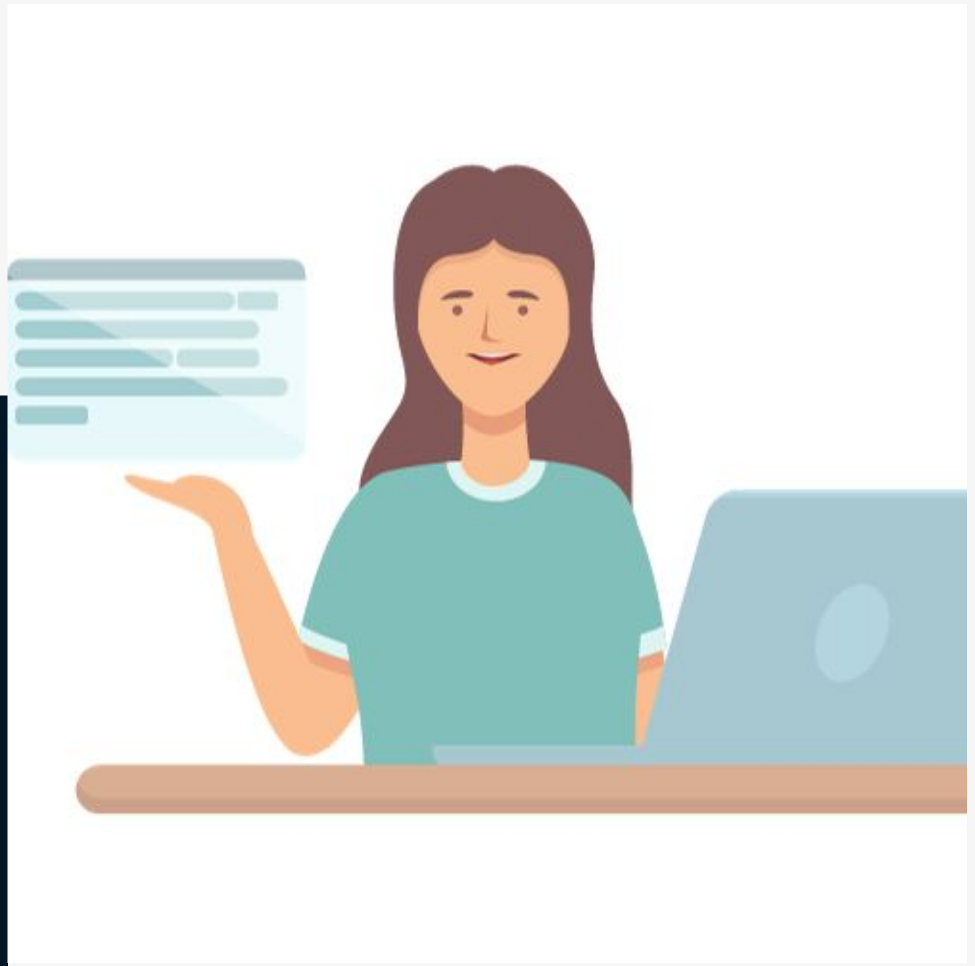
```
class Child(Parent):
    sport = "Tennis"

    def play_sport(self):
        print(f"I play {self.sport}")
```

❖ BOOLEAN

# Using the classes



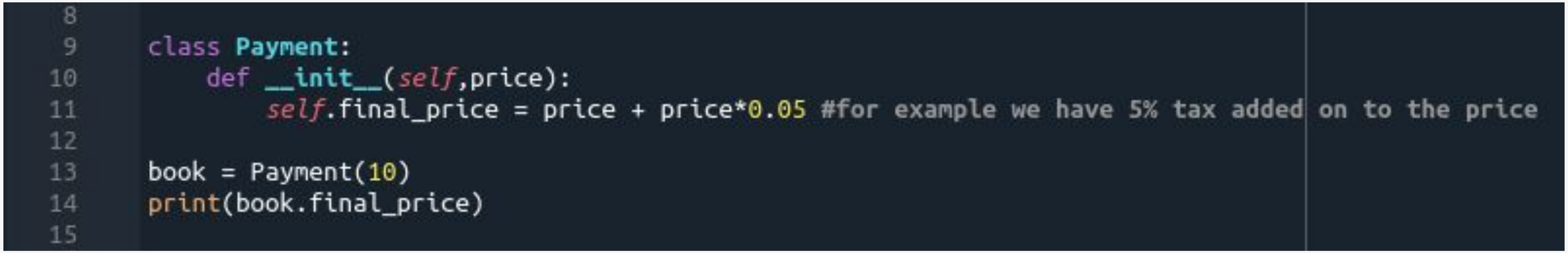
```
child_instance = Child()
child_instance.speak()  # Output: "I speak English"
child_instance.play_instrument()  # Output: "I play the Guitar"
child_instance.play_sport()  # Output: "I play Tennis"
```



3.d Polymorphism
3.e Encapsolation

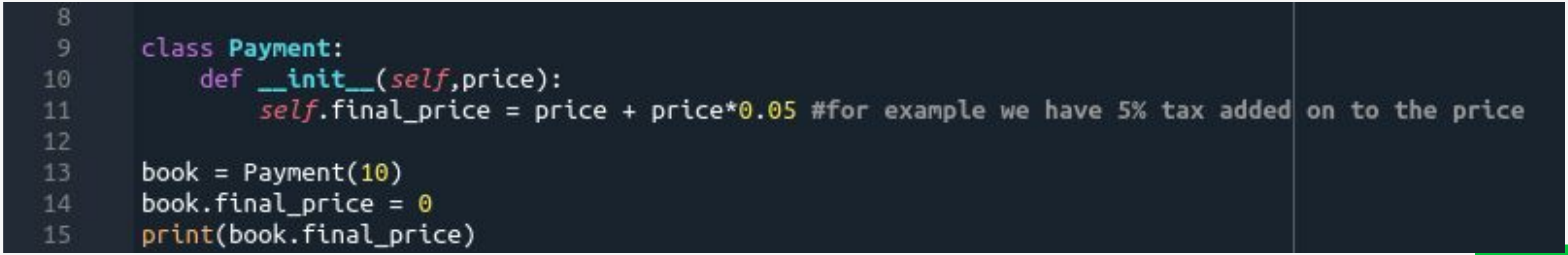
BOOLEAN

OOP Encapsulation Example



```python

class Payment:
    def __init__(self,price):
        self.final_price = price + price*0.05 #for example we have 5% tax added on to the price

book = Payment(10)
print(book.final_price)

```

Output → 10.5

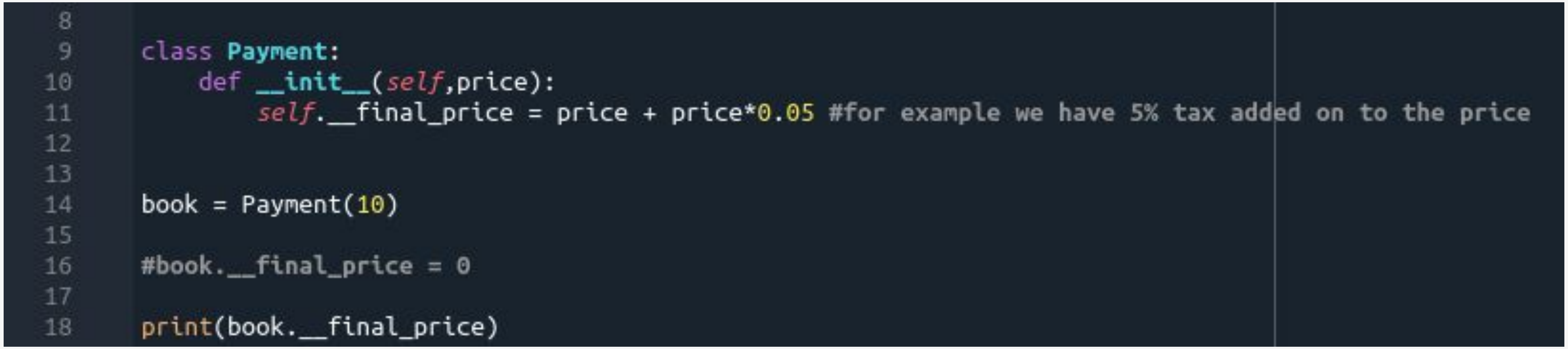


```python

class Payment:
    def __init__(self,price):
        self.final_price = price + price*0.05 #for example we have 5% tax added on to the price

book = Payment(10)
book.final_price = 0
print(book.final_price)
```

Output → 0

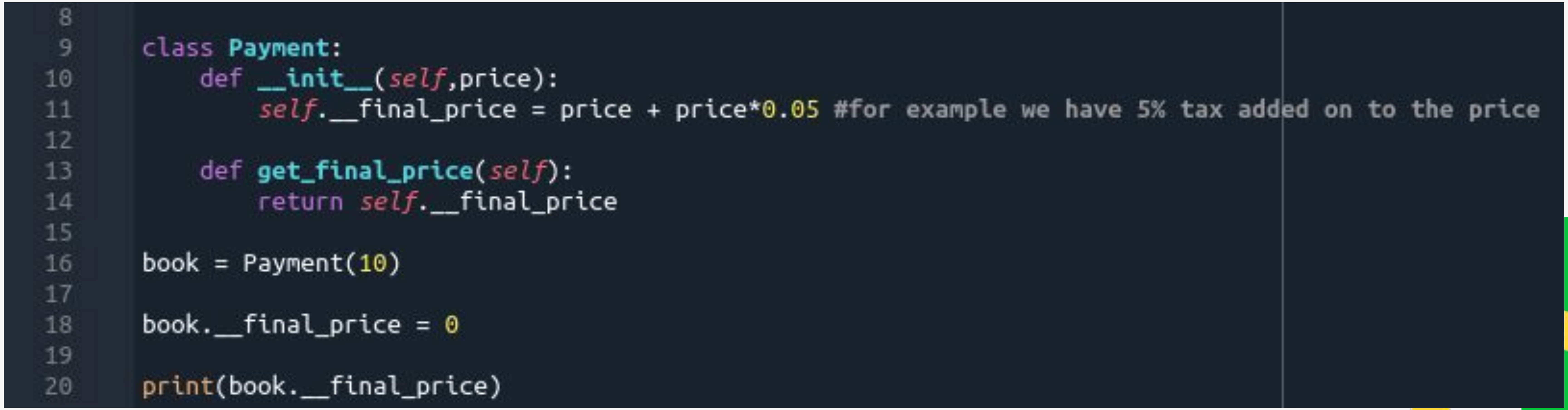
How can we restrict the user from doing such changes?

OOP Encapsulation Example



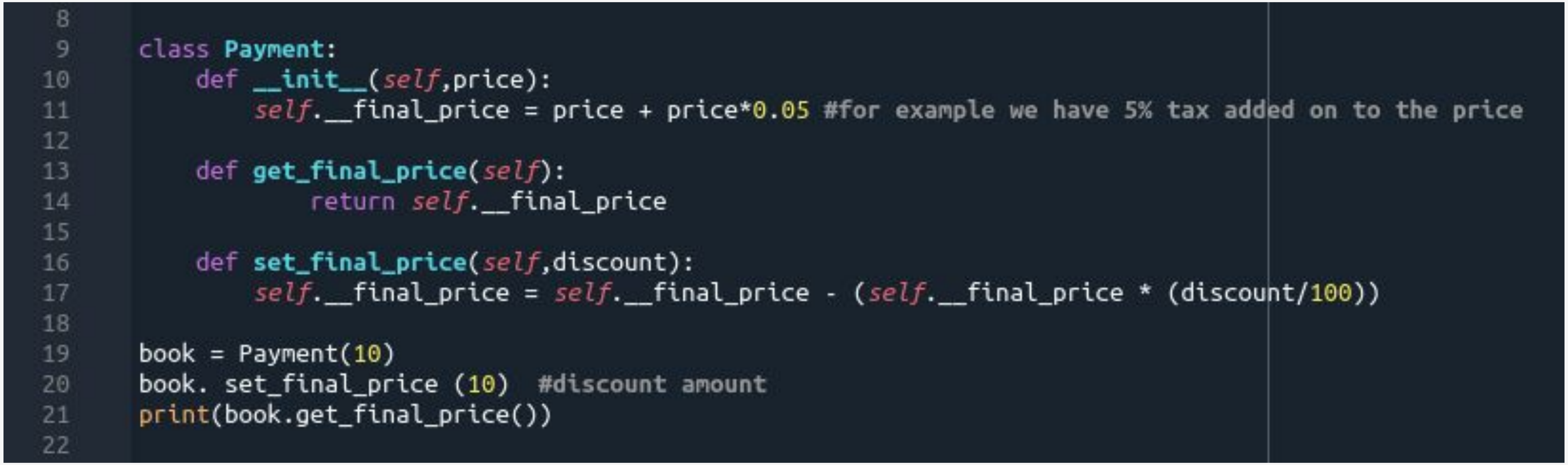
```python

class Payment:
    def __init__(self,price):
        self.__final_price = price + price*0.05 #for example we have 5% tax added on to the price


book = Payment(10)

#book.__final_price = 0

print(book.__final_price)
```

Output → IndentationError: unexpected indent



```python

class Payment:
    def __init__(self,price):
        self.__final_price = price + price*0.05 #for example we have 5% tax added on to the price

    def get_final_price(self):
        return self.__final_price

book = Payment(10)

book.__final_price = 0

print(book.__final_price)
```

OOP Encapsulation Example



```python

class Payment:
    def __init__(self,price):
        self.__final_price = price + price*0.05 #for example we have 5% tax added on to the price

    def get_final_price(self):
            return self.__final_price

    def set_final_price(self,discount):
        self.__final_price = self.__final_price - (self.__final_price * (discount/100))

book = Payment(10)
book. set_final_price (10)  #discount amount
print(book.get_final_price())

```

Output → 9.45

Encapsulation is bundling of attributes and methods inside a single class and being able to make them public or private.

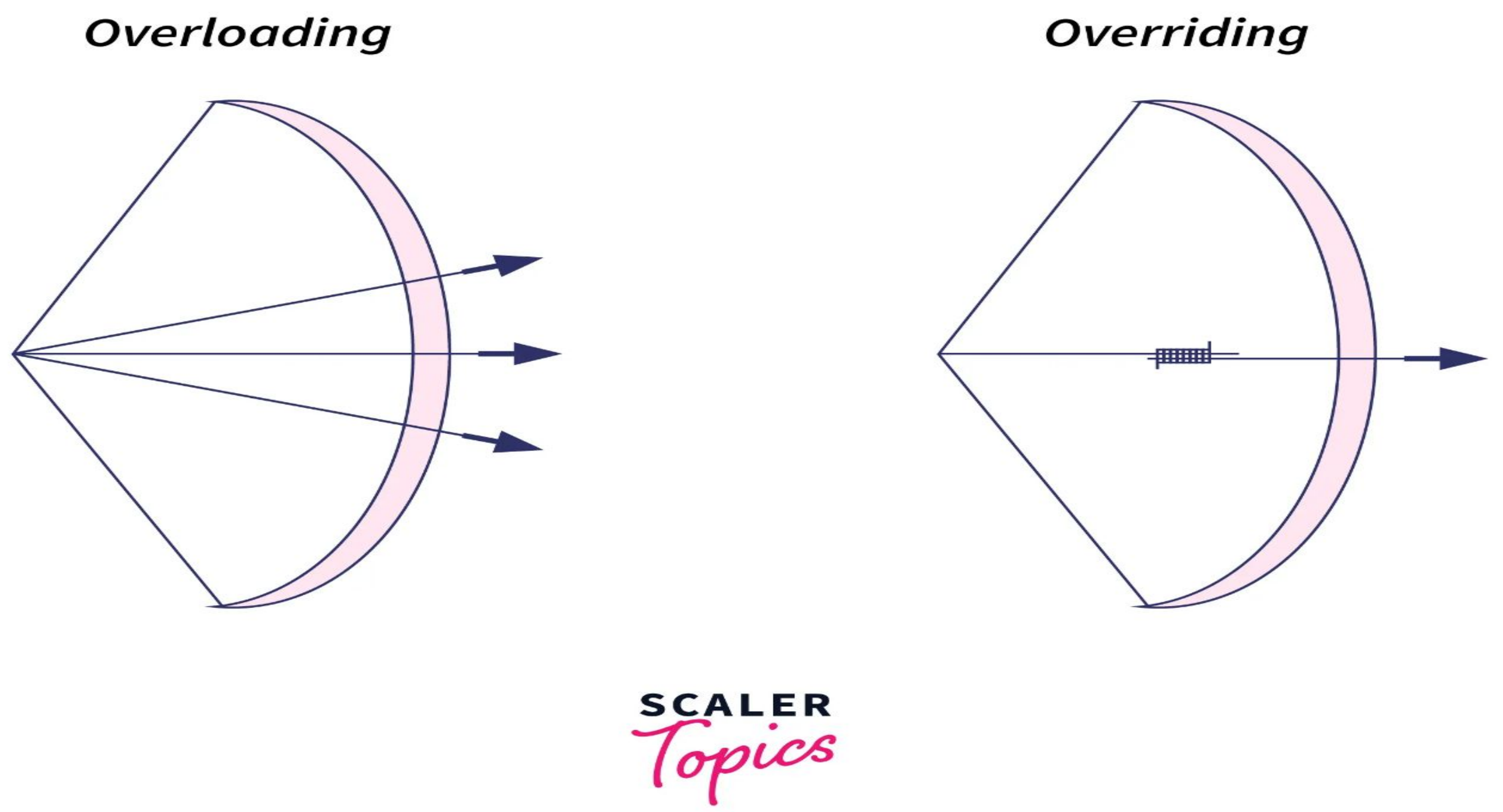
BOOLEAN

3.f Abstraction

BOOLEAN

3.g Overloading/ Overriding

Overloading and **Overriding** in Python are the two main object-oriented concepts that allow programmers to write methods that can process a variety of different types of functionalities with the same name. This helps us to implement Polymorphism and achieve consistency in our code. Let's look at an example to understand the concept of overloading and overriding:
Consider a scenario where you must select arrows for shooting practice in archery. Here, the concept of overloading and overriding can be showed as follows:



First case:
Consider a case where you have 3 similar-looking arrows with different functionalities or properties. Now, depending on the condition of shooting, the arrow is selected. This is known as overloading.
Hence, overloading implies using numerous methods with the same name but accepting a different number of arguments.

Second case:
Now, consider another case where your dad has an arrow. You have access to this arrow, as you inherited it. But, you also bought the same type of arrow for yourself. Now, while shooting if you are choosing your arrow over your dad's arrow, then it is known as overriding.
Now, let's look at overloading and overriding in Python from a technical point of view.

Method Overloading in Python

Method Overloading in Python is a type of <u>Compile-time Polymorphism</u> using which we can <u>define two or more methods in the same class with the same name but with a different parameter list</u>.
We cannot perform method overloading in the Python programming language as everything is considered an object in Python. Out of all the definitions with the same name, it uses the latest definition for the method. Hence, we can define numerous methods with the same name, but we can only use the latest defined method.
In Python, we can make our code have the same features as overloaded functions by defining a method in such a way that there exists more than one way to call it.

For example:

```python
# Function to take multiple arguments
def sum_number(*args):
    # variable to store the sum of numbers
    result = 0
    # accessing the arguments
    for num in args:
        result += num
    # Output
    print("Sum : ", result)
# Driver Code
if(__name__ == "__main__"):
    print("Similar to Method Overloading\n")
    print("Single Argument    ->", end = " ")
    sum_number(10)
    print("Two Arguments      ->", end = " ")
    sum_number(30, 2)
    print("Multiple Arguments ->", end = " ")
    sum_number(1, 2, 3, 4, 5)
```

Code:

Output:

```
Similar to the Method of Overloading

Single Argument     -> Sum :  10
Two Arguments       -> Sum :  32
Multiple Arguments -> Sum :  15
```

Method Overriding in Python

Method Overriding is a type of Run-time Polymorphism. A child class method overrides (or provides its implementation) the parent class method of the same name, parameters, and return type. It is used to over-write (redefine) a parent class method in the derived class.

or example:

```
 Parent Class
lass A:
    def first(self):
        print("First function of class A")

    def second(self):
        print('Second function of class A')

 Derived Class
lass B(A):
    # Overriden Function
    def first(self):
        print("Redefined function of class A in class B")

    def display(self):
        print('Display Function of Child class')

 Driver Code
f(__name__ == "__main__"):
    # Creating child class object
    child_obj = B()

    # Calling the overridden method
    print("Method Overriding\n")
    child_obj.first()

    # Calling the original Parent class method
    # Using parent class object.
    A().first()
```

Output

```
Method Overriding

The redefined function of class A in class B.
The first function of class A.
```

Difference between Overloading and Overriding

Overloading occurs when two or more methods in one class have the same method name but different parameters.

Overriding occurs when two methods have the same method name and parameters.

One of the methods is in the parent class, and the other is in the child class.

When two or more methods in the same class have the same method name but different parameters, this is called overloading.

In contrast, overriding occurs when two methods have the same name and parameters.

INGOUDE COMPANY

Thank You For Listening!

REACH OUT FOR ANY QUESTIONS.

Exploring Abstraction in Object-Oriented Programming

Object-Oriented Programming (OOP) is about breaking down complex systems into manageable parts. Abstraction helps to simplify further by prioritizing key elements.

What is Abstraction?

1

The Concept

In programming, abstraction is the process of removing the complexities of a system by focusing on essential elements and hiding unnecessary details.

2

The Practice

With OOP, abstraction involves creating abstract concepts or models that represent real-world ideas or processes. These abstractions can be used to simplify and organise code.

3

The Benefits

Abstraction makes code more modular, reusable, and scalable. It also helps to reduce the risk of errors and improve maintainability, as changes can be made to abstracted components without affecting the rest of the program.

Applying Abstraction in OOP

Polymorphism

Polymorphism allows objects to take on multiple forms. For example, a vehicle dashboard can be used in various types of vehicles, yet often only needs minor modifications to function optimally

Inheritance

Inheritance enables objects to inherit properties and methods from their parent classes. Much like how a sculptor can create a series of shapes from one block of clay.

Interfaces

Interfaces specify a set of methods that a class must implement. This is akin to a musical performance, where each instrument has its unique role to create harmony together.

Abstraction vs Encapsulation

Abstraction

Focuses on the essential features of a system, simplifies code, and hides unnecessary details.

Encapsulation

Protects data and functionality within a class from other classes, limits access to key elements, and ensures data integrity

When to Use Abstraction?

1

Complex Systems

When dealing with complex systems, abstraction can help to simplify logic and reduce code duplication.

2

Large Codebases

In large programs, abstraction can improve maintainability by separating components into manageable modules.

3

Collaborative Projects

Within collaborative projects, abstraction can assist in creating consistent coding standards and ensure code is more reusable.

Summary and Conclusion

1 The Key Takeaways

Abstraction is the process of removing complexities of a system, it enables code to be modular, reusable, and scalable, and there are three significant forms of abstraction in OOP: Polymorphism, Inheritance, and Interfaces.

2 The Benefits

Abstraction helps programmers to reduce code complexity, make changes more easily and catch errors earlier, and ultimately deliver better software faster.

3 The Importance

Abstraction is fundamental to the logic of programming and is one of the essential tools available to programmers to be able to tackle problems effectively.

Abstraction Example

Abstract Classes and Methods: In Python, you can create an abstract class using the ‘`abc`’ module. Abstract classes can have abstract methods, which are defined in the base class but must be implemented in derived classes.

In this example, the ‘`Shape`’ class is an abstract class with an abstract method ‘`area()`’. Concrete classes like ‘`Circle`’ and ‘`Rectangle`’ must implement the ‘`area`’ method, enforcing abstraction.

```
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 3.14 * self.radius * self.radius

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("Circle Area:", circle.area())
print("Rectangle Area:", rectangle.area())
```