Artificial Intelligence Diploma

Python Session 8

NeuroTech

Agenda

1. Inheritance
2. Type of Inheritance
3. Encapsulation
4. Abstraction
5. Polymorphism

Inheritance

NeuroTech

Inheritance

- Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a new class (subclass or derived class) to inherit properties (attributes and methods) from an existing class (base class or superclass). Inheritance establishes a parent-child relationship between classes, where the subclass inherits the attributes and methods of the superclass.

- The concept of inheritance is analogous to inheriting traits or characteristics from one's parents. In the context of programming, it provides a way to create new classes that are based on existing classes, allowing for code reuse, modularity, and the extension of functionality.

- Inheritance offers several benefits in object-oriented programming, contributing to code organization, reusability, and the creation of efficient and maintainable software.

Inheritance

- **Some of the key benefits of using inheritance:**

1. **Code Reusability:** Inheritance allows you to define common attributes and methods in a base class (superclass) and reuse them in multiple subclasses. This reduces code duplication and promotes the DRY (Don't Repeat Yourself) principle, leading to more efficient development and easier maintenance.

2. **Modularity:** Inheritance encourages the creation of modular code. You can encapsulate common behaviors in the base class and extend or customize those behaviors in subclasses. This improves code organization and makes it easier to manage and understand the codebase.

3. **Hierarchical Organization:** Inheritance facilitates the creation of class hierarchies that model real-world relationships. This hierarchical structure enhances the clarity of your code, as it reflects the natural relationships between different classes.

4. **Efficiency:** Inherited methods and attributes are reused, which can lead to more efficient memory usage and optimized execution. You're not creating redundant methods for similar functionality.

Inheritance

6. Polymorphism: Inheritance enables polymorphism, where objects of different classes can be treated as objects of a common superclass. This promotes flexibility and allows you to write more generic code that can work with different types of objects.

7. Maintenance: Changes made to the base class are automatically reflected in all subclasses, ensuring consistency and reducing the likelihood of errors. This simplifies maintenance tasks and updates across the codebase.

8. Code Extensibility: As your application evolves, you can add new subclasses to extend the functionality of the existing classes. This promotes extensibility without affecting the existing code.

9. Abstraction: Inheritance allows you to create abstract classes that define common attributes and methods without providing full implementations. This enforces a consistent structure and behavior across subclasses.

Inheritance

10. Simplifying Complex Systems: In large and complex systems, inheritance can help break down the functionality into manageable and logical components. Each subclass can focus on a specific aspect of the overall system.

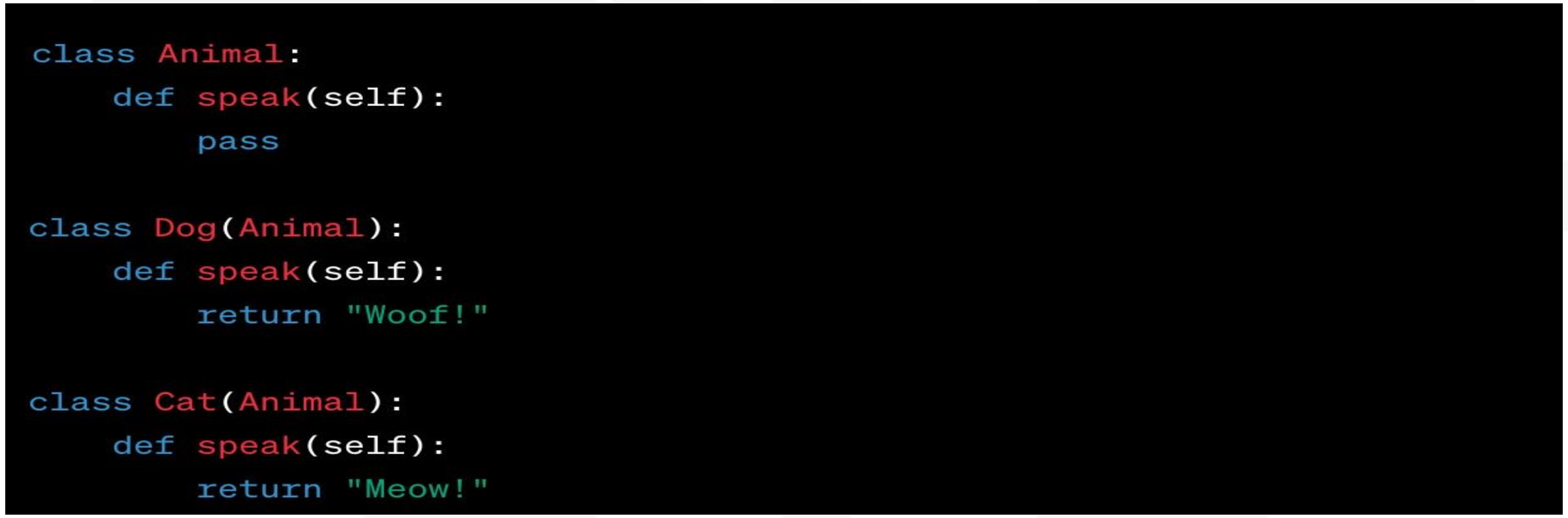
11. Reduced Development Time: By reusing existing code through inheritance, you can save development time. You're building on a foundation of established functionality, allowing you to concentrate on the unique aspects of your subclasses.

Type of inheritance

1. Single Inheritance.
2. Multiple Inheritance.
3. Multilevel Inheritance.
4. Hierarchical Inheritance.
5. Hybrid Inheritance.

Single Inheritance

- In single inheritance, a subclass inherits from a single superclass. This is the simplest form of inheritance.



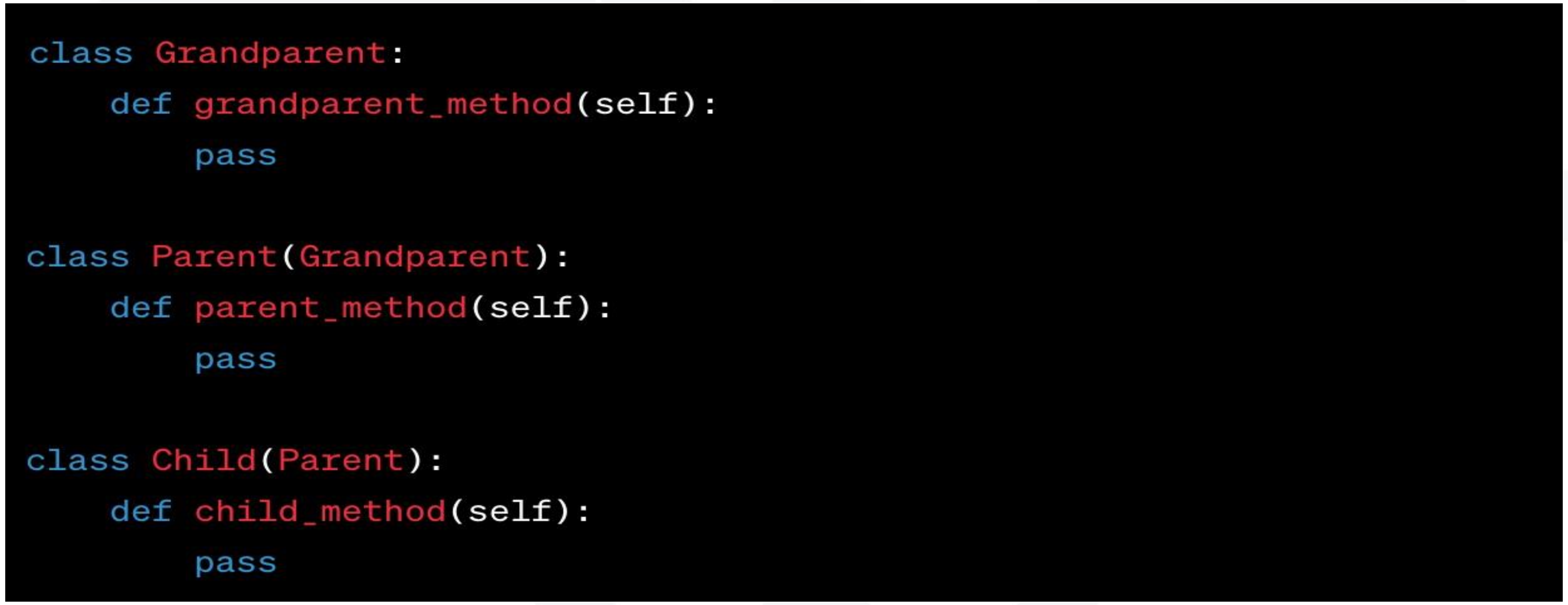
```
class Animal:
    def speak(self):
        pass

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

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

- In this example, both Dog and Cat inherit from the Animal class. Each subclass provides its own implementation of the speak() method.

NeuroTech

Multiple Inheritance

- In multiple inheritance, a subclass inherits from more than one superclass. This allows a class to inherit attributes and methods from multiple parent classes.



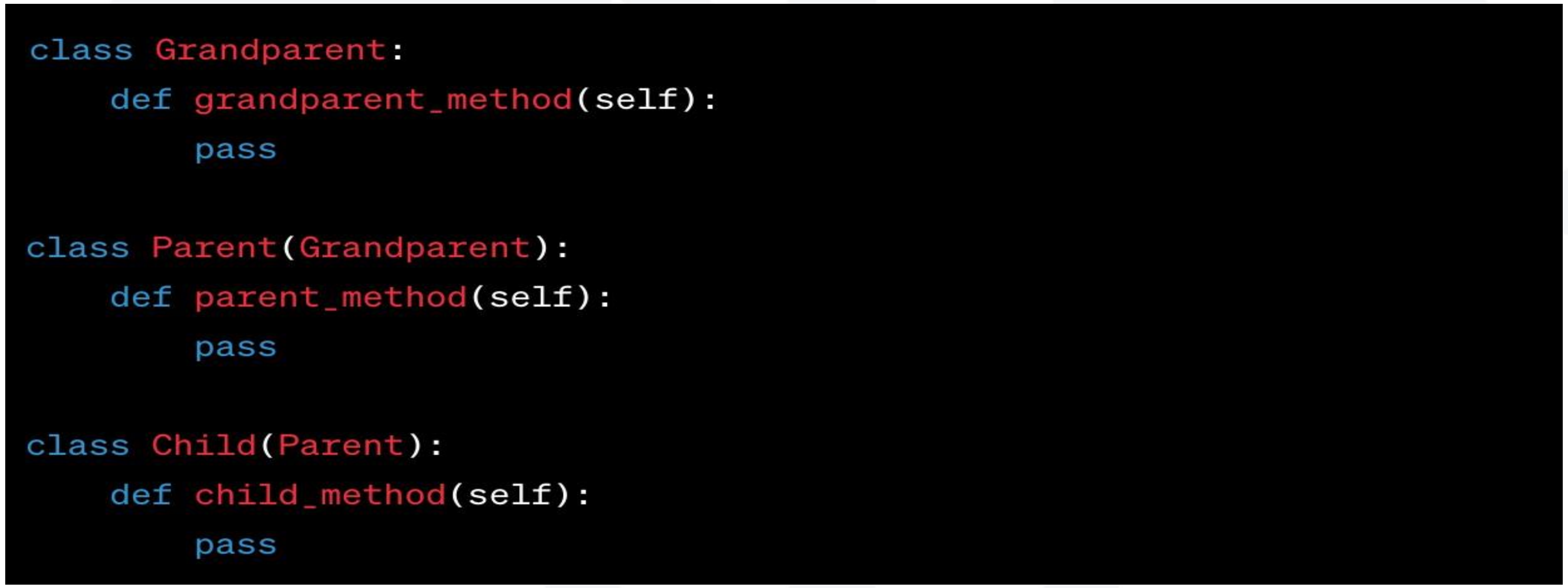
```
class Grandparent:
    def grandparent_method(self):
        pass

class Parent(Grandparent):
    def parent_method(self):
        pass

class Child(Parent):
    def child_method(self):
        pass
```

- Here, the Car class inherits from both Vehicle and Engine classes.

NeuroTech

Multilevel Inheritance

- In multilevel inheritance, a chain of inheritance is created where a subclass becomes the superclass for another subclass.



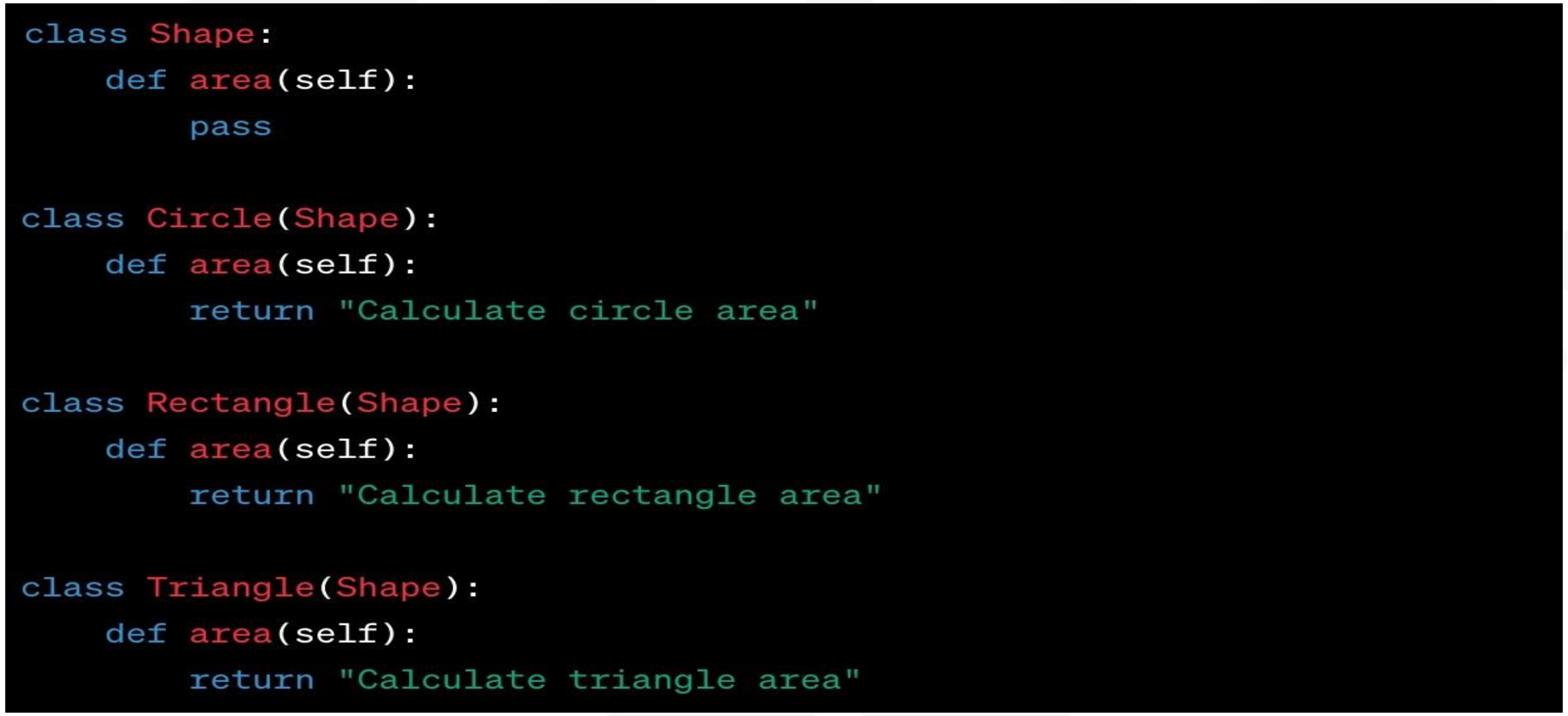
```python
class Grandparent:
    def grandparent_method(self):
        pass

class Parent(Grandparent):
    def parent_method(self):
        pass

class Child(Parent):
    def child_method(self):
        pass
```

- Here, Child inherits from Parent, which in turn inherits from Grandparent.

NEUROTECH

Hierarchical Inheritance

- In hierarchical inheritance, multiple subclasses inherit from a single superclass.

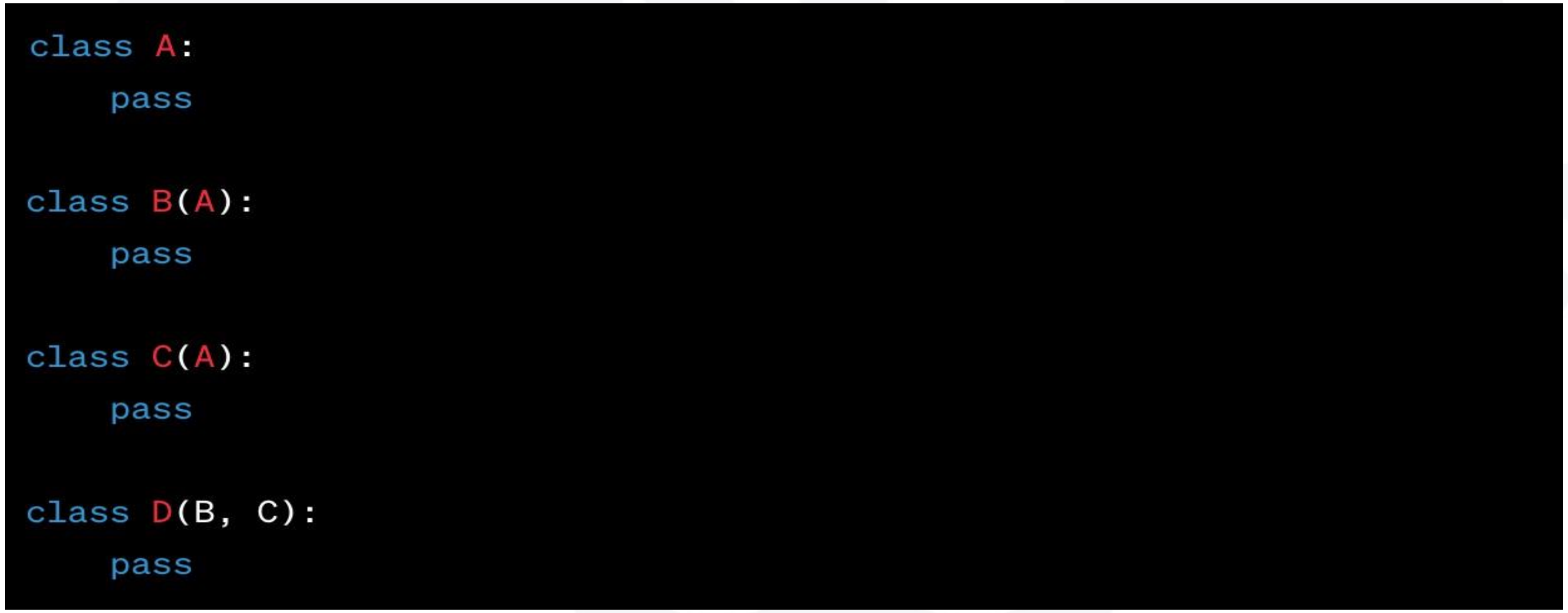


```
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def area(self):
        return "Calculate circle area"

class Rectangle(Shape):
    def area(self):
        return "Calculate rectangle area"

class Triangle(Shape):
    def area(self):
        return "Calculate triangle area"
```

- All three subclasses (Circle, Rectangle, Triangle) inherit from the Shape superclass.

Hybrid Inheritance

- Hybrid inheritance involves a combination of multiple inheritance types. It can lead to complex class hierarchies.



```python
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass
```

- In this example, class D inherits from both B and C, resulting in hybrid inheritance.

Encapsulation

Encapsulation

- Encapsulation is one of the fundamental concepts of Object-Oriented Programming (OOP) that focuses on bundling data (attributes) and the methods (functions) that operate on that data into a single unit called a class. Encapsulation helps in restricting direct access to some of the object's components, providing control over how data is accessed and modified. This concept is often summarized with the phrase "data hiding.

Key Concepts of Encapsulation:

1. **Access Modifiers:**
    1. Access modifiers (also known as access specifiers) determine the visibility and accessibility of class members (attributes and methods).
    2. Common access modifiers include public, protected, and private.
2. **Private Attributes and Methods:**
    1. Private attributes and methods are marked as private and are intended to be used only within the class itself.
    2. They are not accessible from outside the class.
3. **Getters and Setters:**
    1. Encapsulation promotes the use of getters (methods to retrieve attribute values) and setters (methods to set attribute values).
    2. Getters and setters provide controlled access to attributes, allowing validation and manipulation of data before it's stored or retrieved.

Encapsulation

1. Access Modifiers:

Python provides three levels of access modifiers to control the visibility of class members (attributes and methods):

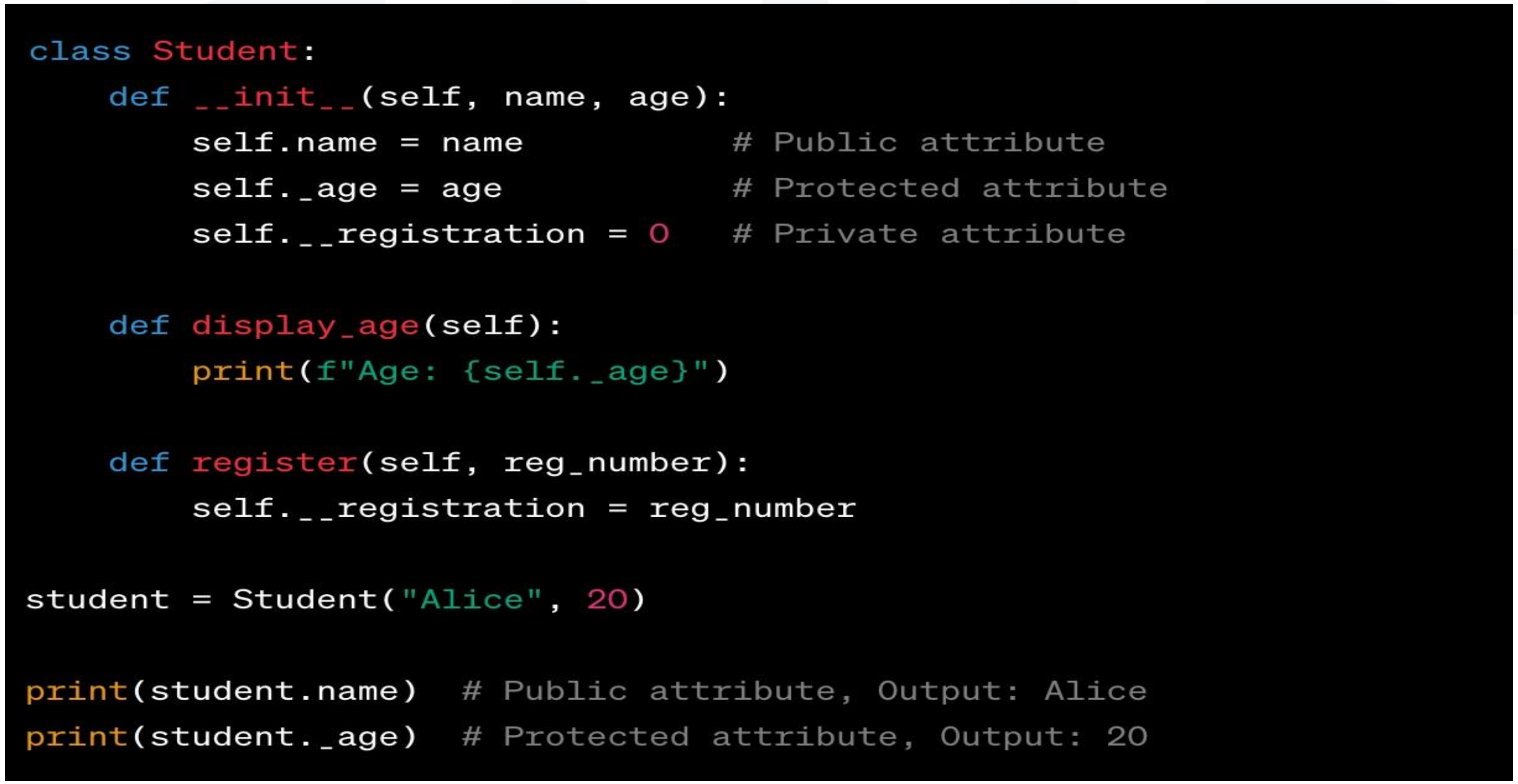
1. **Public:** Members with no access modifier are considered public and can be accessed from anywhere.
2. **Protected:** Members prefixed with a single underscore (e.g., _attribute) are considered protected. They can be accessed within the class and its subclasses.
3. **Private:** Members prefixed with double underscores (e.g.,___attribute) are considered private. They can only be accessed within the class itself.

Getter and Setter Methods:

Getter and setter methods provide controlled access to attributes, allowing validation and manipulation before setting or retrieving values.

Encapsulation

Example: Encapsulation with Access Modifiers



```python
class Student:
    def __init__(self, name, age):
        self.name = name            # Public attribute
        self._age = age             # Protected attribute
        self.__registration = 0    # Private attribute

    def display_age(self):
        print(f"Age: {self._age}")

    def register(self, reg_number):
        self.__registration = reg_number

student = Student("Alice", 20)

print(student.name)  # Public attribute, Output: Alice
print(student._age)  # Protected attribute, Output: 20
```

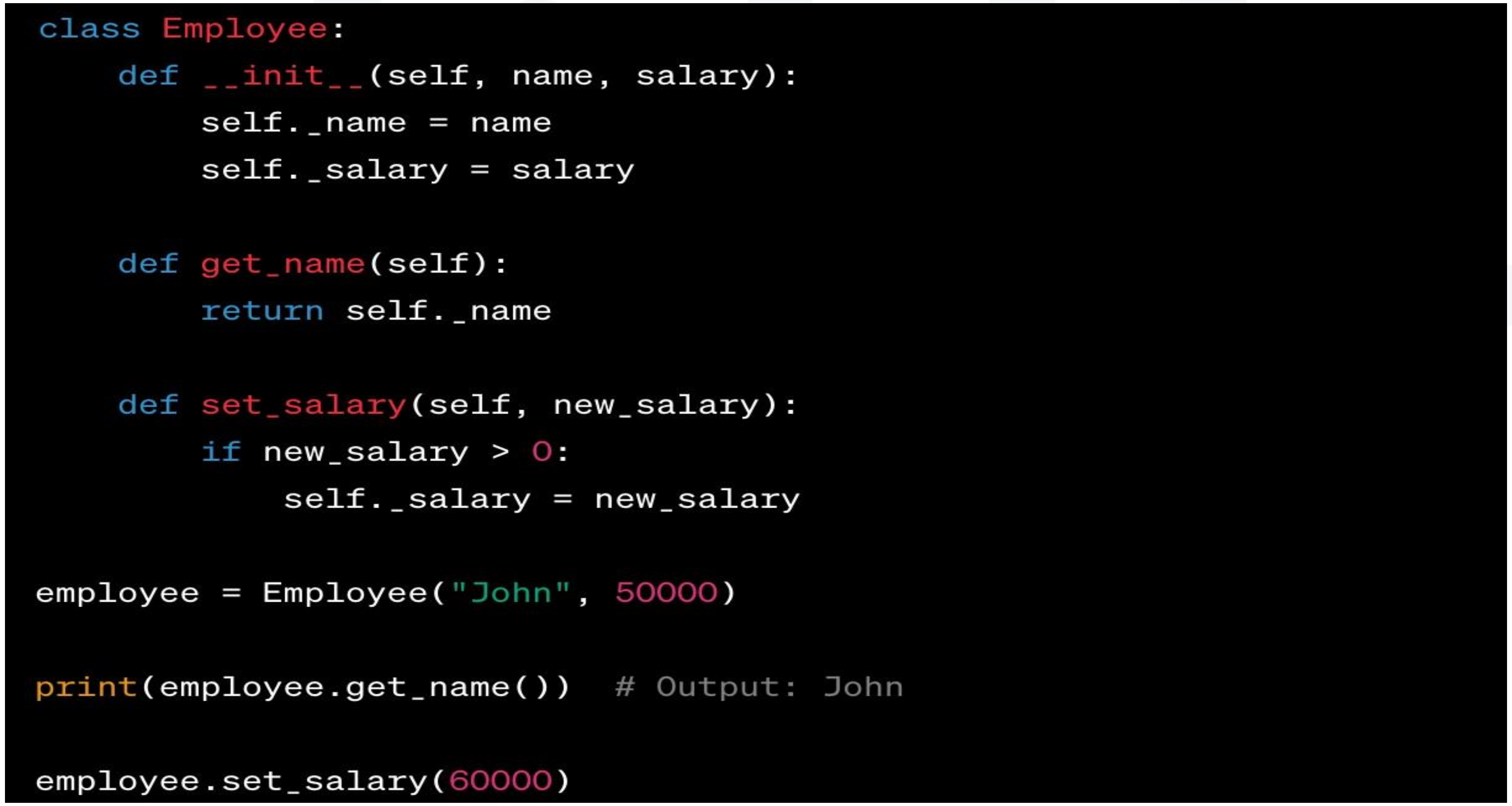
NEUROTECH

Encapsulation

In previous example, name is public, _age is protected, and___registration is private. While the private attribute can be accessed using name mangling (_Student__registration), it's generally not recommended to directly access private attributes from outside the class.

Encapsulation

Getter and Setter:



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

    def get_name(self):
        return self._name

    def set_salary(self, new_salary):
        if new_salary > 0:
            self._salary = new_salary

employee = Employee("John", 50000)

print(employee.get_name())  # Output: John

employee.set_salary(60000)
```

NEUROTECH

Encapsulation

Benefits of Encapsulation:

1. **Data Protection:** Encapsulation prevents direct access to internal attributes, reducing the risk of unauthorized changes and ensuring data integrity.
2. **Code Maintenance:** Encapsulation allows you to change the internal implementation of a class without affecting external code that uses the class.
3. **Abstraction:** Encapsulation enables you to expose a simplified interface to users while hiding complex implementation details.
4. **Controlled Access:** Using getter and setter methods, you can control how attributes are accessed and modified, enforcing validation rules.
5. **Modularity:** Encapsulation promotes modular design by bundling related attributes and methods within a class.

Abstraction

NeuroTech

Abstraction

- Abstraction is one of the core principles of Object-Oriented Programming (OOP), aiming to simplify complex systems by modeling classes based on essential attributes and behaviors. It involves creating a blueprint for objects without exposing all the implementation details. Abstraction allows programmers to focus on what an object does rather than how it does it.

How Abstraction Works:

1. **Identify Essential Features:** In the process of abstraction, you identify the most important attributes and methods that define an object's behavior.
2. **Ignore Implementation Details:** Abstraction encourages you to ignore the internal implementation complexities of an object. Instead, you focus on providing a clear interface for interacting with the object.
3. **Create a Class:** The identified attributes and methods are then encapsulated within a class. The class defines the abstract representation of the object, including what it can do and what data it holds.
4. **Hide Complexity:** Abstraction hides unnecessary details and complexities, making the object easier to understand and use.

Abstraction

Example of Abstraction:

Consider a real-world example of a remote control for a television. When you use a remote control, you don't need to know how the remote communicates with the TV or how the TV processes the signals. You only need to know the buttons to change channels, adjust volume, and turn the TV on/off.

In terms of abstraction:

- The remote control is an abstraction of the TV control interface.
- You interact with the remote's buttons (methods) to perform actions (turn on/off, change channels).
- The internal details of how the remote works or how the TV responds are abstracted away.

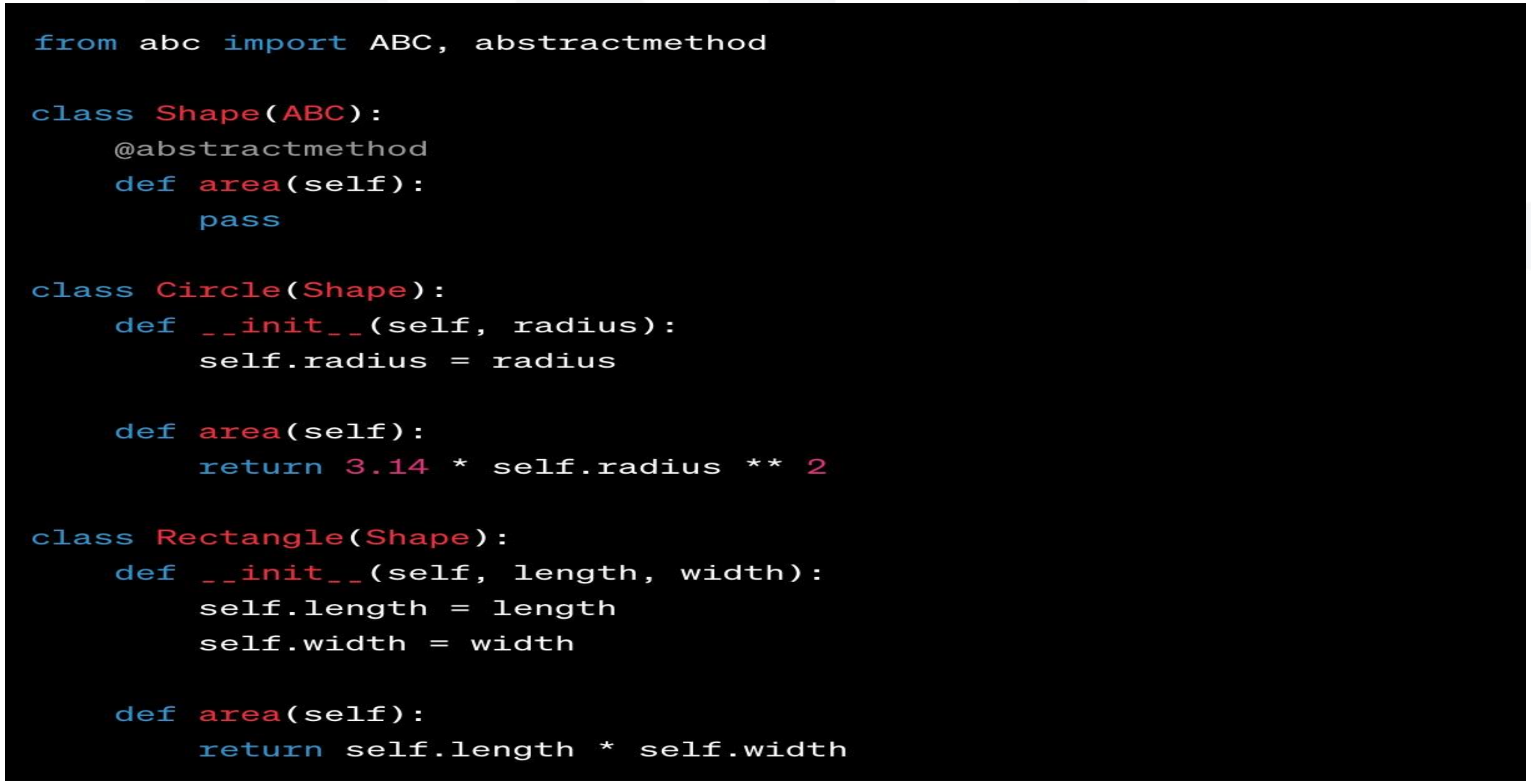
Abstraction

Benefits of Abstraction:

1. **Simplification:** Abstraction simplifies the complex real world by focusing on the most important aspects.
2. **Modularity:** Abstraction allows you to create modular components with clear interfaces.
3. **Code Reusability:** Abstract classes can be inherited to create new classes, promoting code reuse.
4. **Encapsulation:** Abstraction enforces the encapsulation of data and behavior within classes.
5. **Flexibility:** You can change the internal implementation of a class without affecting external code.

Abstraction

Python provides the abc (Abstract Base Classes) module for creating abstract classes. An abstract class cannot be instantiated and can have abstract methods that must be implemented by its subclasses.



```python
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 ** 2

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

    def area(self):
        return self.length * self.width
```

NeuroTech

Abstraction

In Previous example, **Shape** is an **abstract class**, and **Circle and Rectangle** are its **concrete subclasses**. Each subclass provides its own implementation of the area() method, adhering to the abstract contract defined by the Shape class.

Abstraction vs. Encapsulation:

While abstraction and encapsulation are related concepts in OOP, they serve slightly different purposes:

- **Abstraction:** Focuses on providing a clear interface and hiding unnecessary implementation details of an object. It defines what the object does.

- **Encapsulation:** Focuses on bundling data and methods that operate on the data into a single unit, a class. It defines how the object works.

Polymorphism

Polymorphism

Polymorphism is a fundamental concept in Object-Oriented Programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent a general class of actions, which can be performed in a similar way regardless of the specific implementation in subclasses.

In simpler terms, polymorphism allows you to write code that can work with objects of different classes in a uniform manner, as long as they share a common interface.

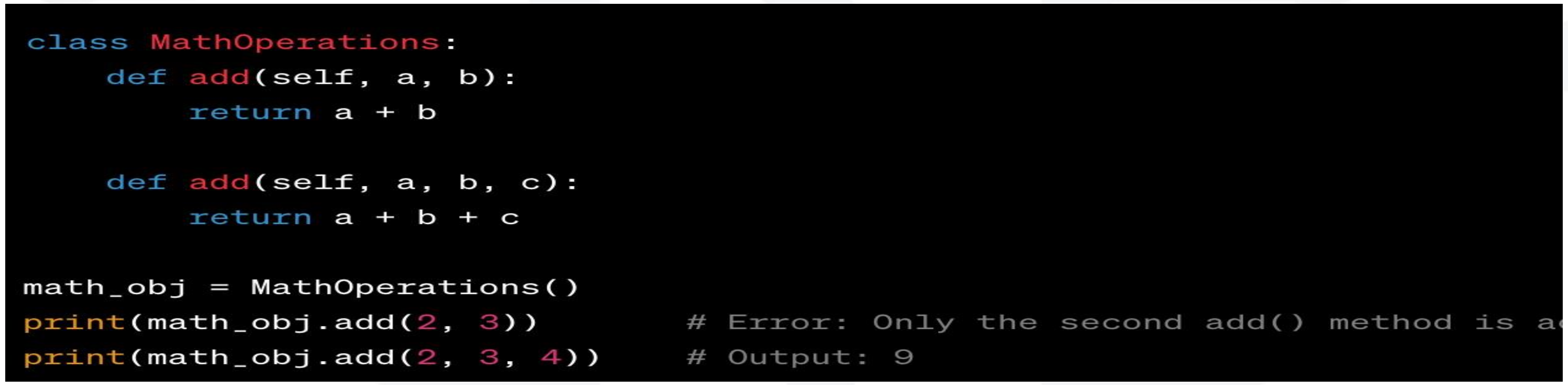
Types of Polymorphism:

1. **Compile-Time (Static) Polymorphism:**
    1. Also known as method overloading or compile-time method resolution.
    2. Occurs when the method to be invoked is determined at compile time based on the method's signature.
    3. Involves the use of multiple methods in the same class with the same name but different parameters.
2. **Run-Time (Dynamic) Polymorphism:**
    1. Also known as method overriding or dynamic method resolution.
    2. Occurs when the method to be invoked is determined at runtime based on the actual object's type.
    3. Involves defining a method in a subclass with the same name, parameters, and return type as a method in its superclass.

Polymorphism

Compile-Time (Static) Polymorphism Example:

Compile-time polymorphism is achieved through method overloading. It allows a class to have multiple methods with the same name but different parameters. The appropriate method to call is determined by the number and types of arguments passed during compilation.



```
class MathOperations:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c

math_obj = MathOperations()
print(math_obj.add(2, 3))        # Error: Only the second add() method is a
print(math_obj.add(2, 3, 4))     # Output: 9
```

In this example, the class MathOperations has two add() methods with different numbers of parameters. However, due to the nature of Python, only the second add() method is accessible, and attempting to call the first add() method will result in an error.

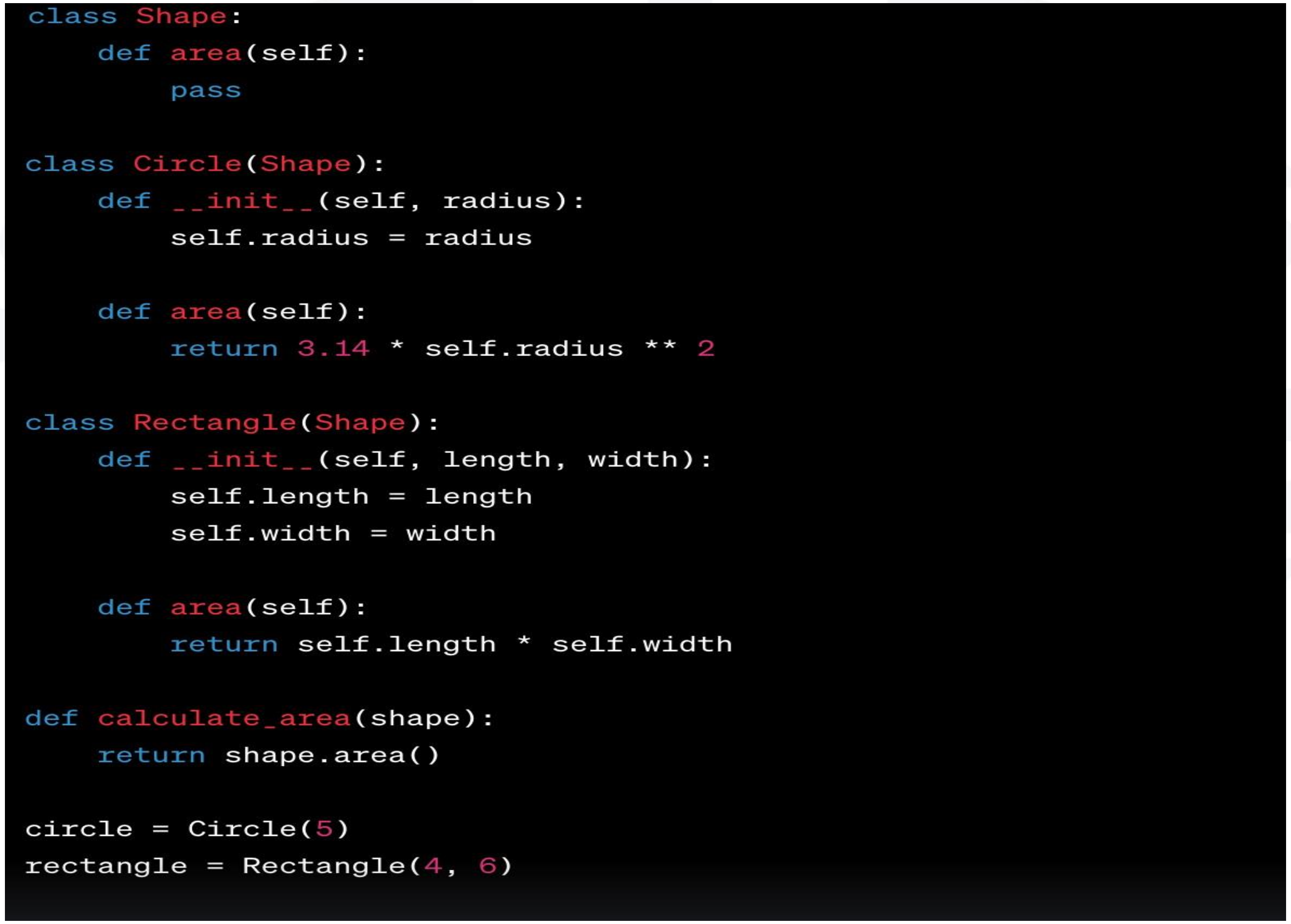
NEUROTECH

Polymorphism

Run-Time (Dynamic) Polymorphism Example:

Run-time polymorphism is achieved through method overriding. It allows a subclass to provide a specific implementation for a method that's already defined in its superclass. The appropriate method to call is determined by the actual object's type during runtime.

Polymorphism



```python
class Shape:
    def area(self):
        pass

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

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

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

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

def calculate_area(shape):
    return shape.area()

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

NEUROTECH

Polymorphism

In this example, the **Shape** class defines an **area()** method, and the **Circle** and **Rectangle** subclasses override this method with their own implementations. The **calculate_area()** function takes any **Shape** object as a parameter and calls its **area()** method. The appropriate **area()** method is determined at runtime based on the object's type.

Polymorphism

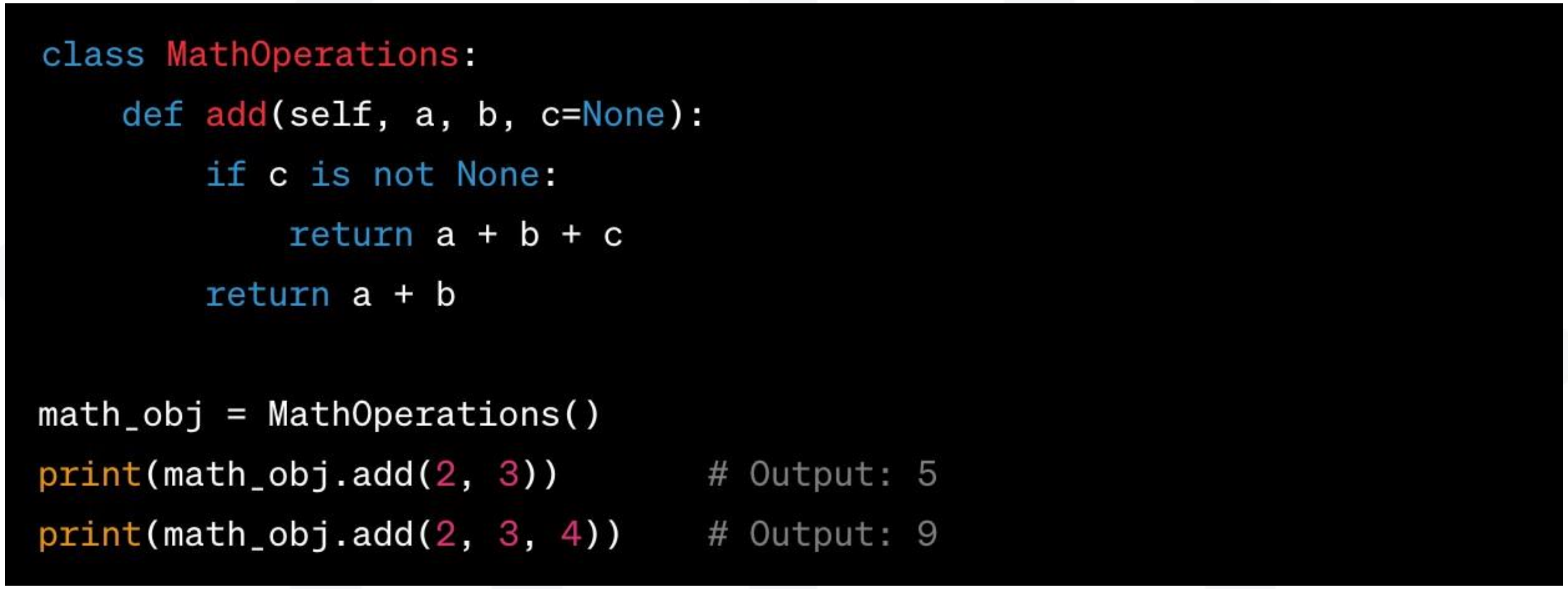
Overloading and **Overriding** are two important concepts in polymorphism within the context of Object-Oriented Programming (OOP). They both enable the manipulation of methods in a way that promotes flexibility and reusability. However, they are used in different scenarios and have distinct purposes.

Method Overloading:

- **Method overloading** is a feature that allows a class to have multiple methods with the same name but different parameters. This provides more flexibility when calling methods, as the appropriate method to call is determined by the number and types of arguments provided. In some programming languages, like Java, method overloading is explicitly supported, but in Python, method overloading is achieved in a slightly different manner.

- In Python, you can't define multiple methods with the same name and different parameter lists like you can in some other languages. However, you can achieve similar behavior using default arguments or variable-length argument lists.

Polymorphism



```
class MathOperations:
    def add(self, a, b, c=None):
        if c is not None:
            return a + b + c
        return a + b

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

In this example, the **add()** method can take two or three arguments. If three arguments are provided, it adds them together. Otherwise, it adds the first two arguments.

Polymorphism

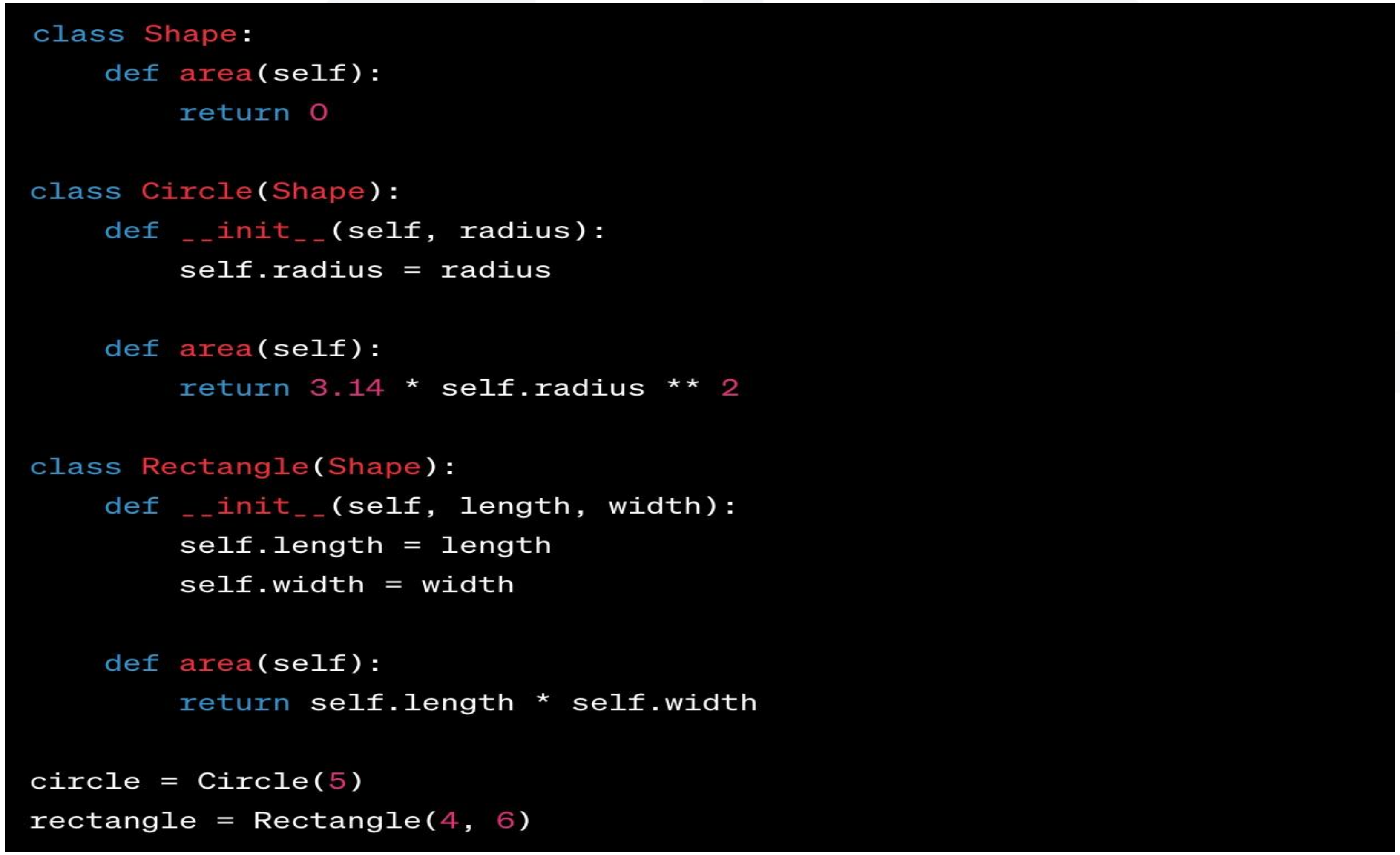
- **Method overriding** is the practice of providing a specific implementation for a method that's already defined in a superclass. It allows subclasses to provide their own version of the method, which is then used instead of the method in the superclass when called on an object of the subclass.

- In next example, the **Circle** and **Rectangle** subclasses override the **area()** method from the Shape superclass. When you call **area()** on an object of a subclass, the appropriate overridden method is called based on the actual object's type.

In summary:

- **Method Overloading** is about defining multiple methods with the same name but different parameter lists to provide more flexibility in calling methods.
- **Method Overriding** is about providing a specific implementation for a method in a subclass that's already defined in a superclass to customize the behavior for the subclass.

Polymorphism



```
class Shape:
    def area(self):
        return 0

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

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

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

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

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

Practice "Go to Notebook”

NeuroTech

Any Question

NeuroTech

Thanks

NeuroTech