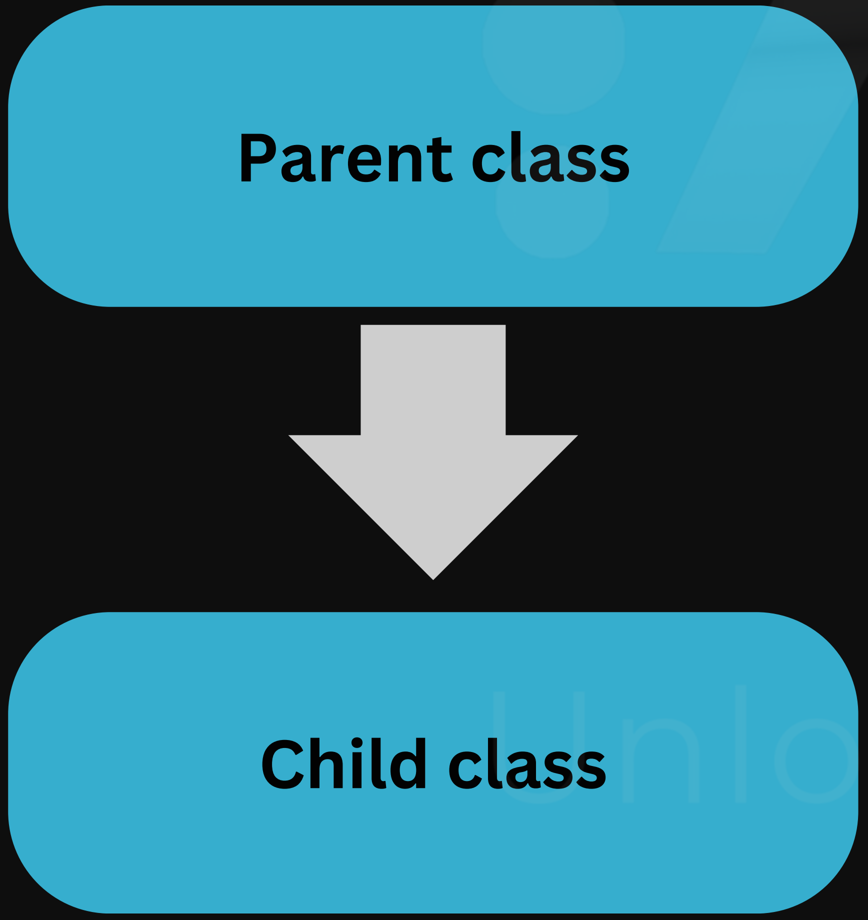
Python Inheritance

Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP) in Python. It allows a class to inherit attributes and methods from another class, promoting code reusability and hierarchy.

What is Inheritance

Inheritance is a mechanism where one class derives properties and behaviors (methods) from another class.



Base class / Superclass
The class whose properties are inherited.

Derived class / Subclass
The class that inherits from another class

Why Use Inheritance ?

- **Code Reusability :**

  Avoids duplication of code.

- **Improves Maintainability :**

  Changes in the parent class reflect in the child class.

- **Encapsulation :**

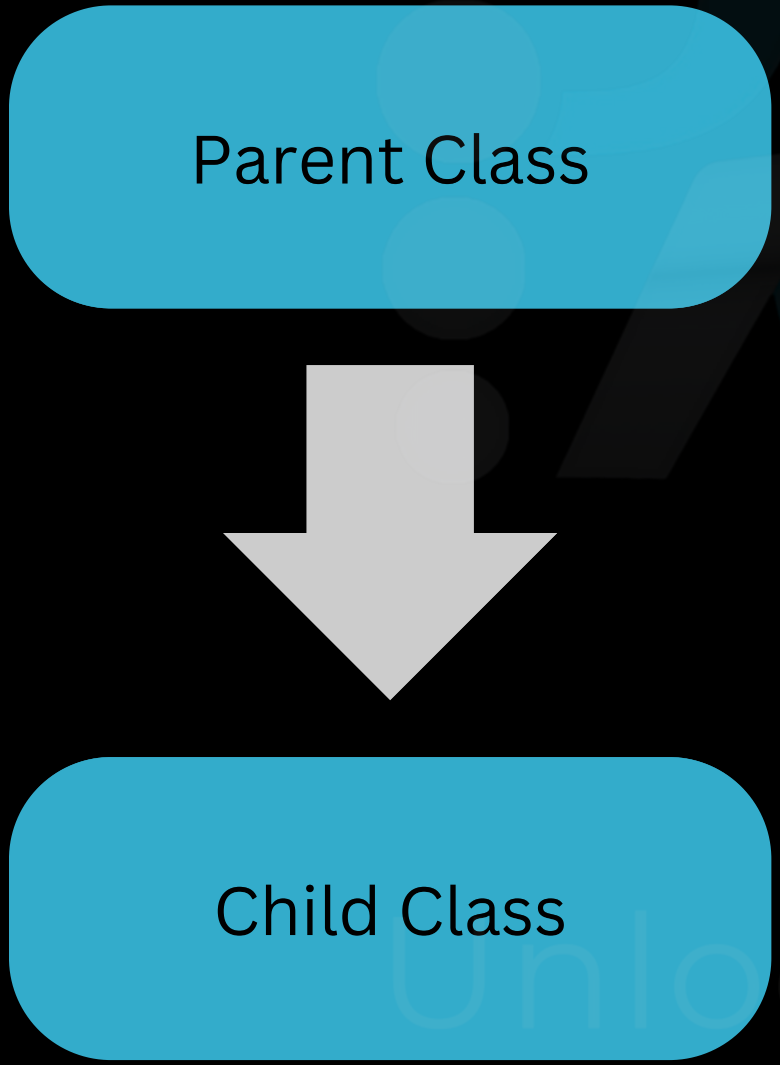
  Allows you to structure your code in a hierarchical way

Types of Inheritance

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance

A child class inherits from a single parent class.

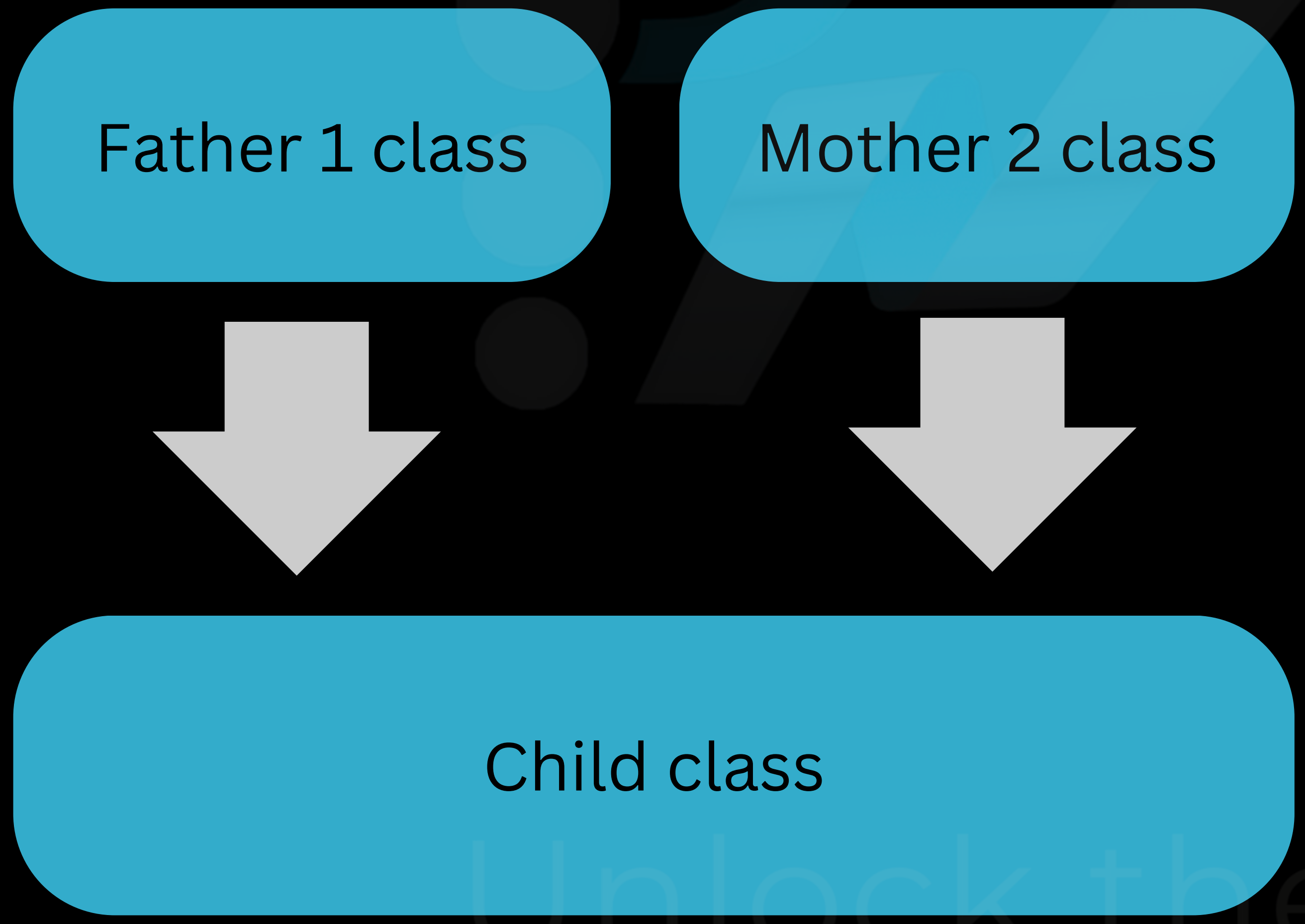


Example

```
class Parent:
    def parent_method(self):
        return "This is parent class"

class Child(Parent):
    def child_method(self):
        return "This is child class"


# Usage
child = Child()
print(child.parent_method())
# Outputs: This is parent class
print(child.child_method())
 # Outputs: This is child class
```

Multiple Inheritance

In multiple inheritance, a child class inherits from more than one parent class.

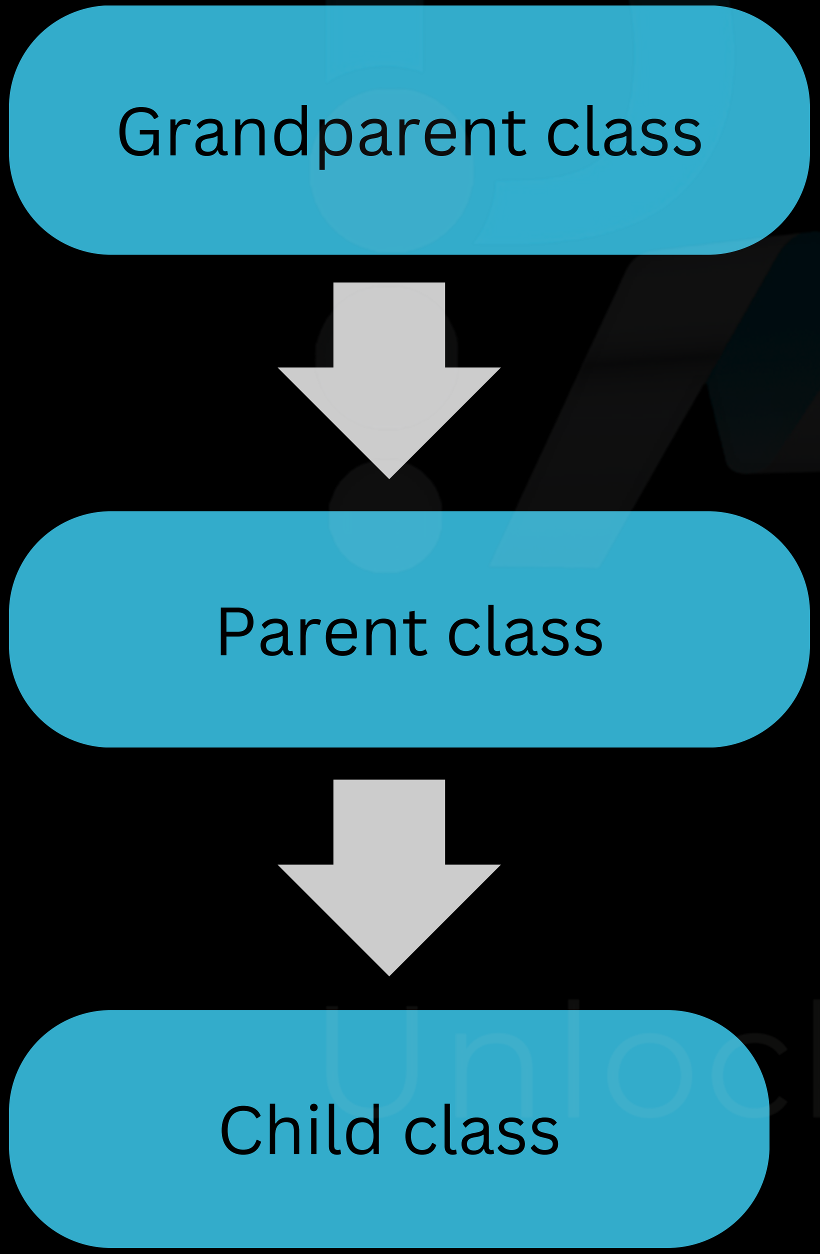


Example

```
class Father:
    def father_method(self):
        return "Father's trait"

class Mother:
    def mother_method(self):
        return "Mother's trait"

class Child(Father, Mother):
    def child_method(self):
        return "Child's trait"


# Usage
child = Child()
print(child.father_method())
# Outputs: Father's trait
print(child.mother_method())
# Outputs: Mother's trait
```

Multilevel Inheritance

In multilevel inheritance, a child class inherits from a parent class, and another child class inherits from that child class.

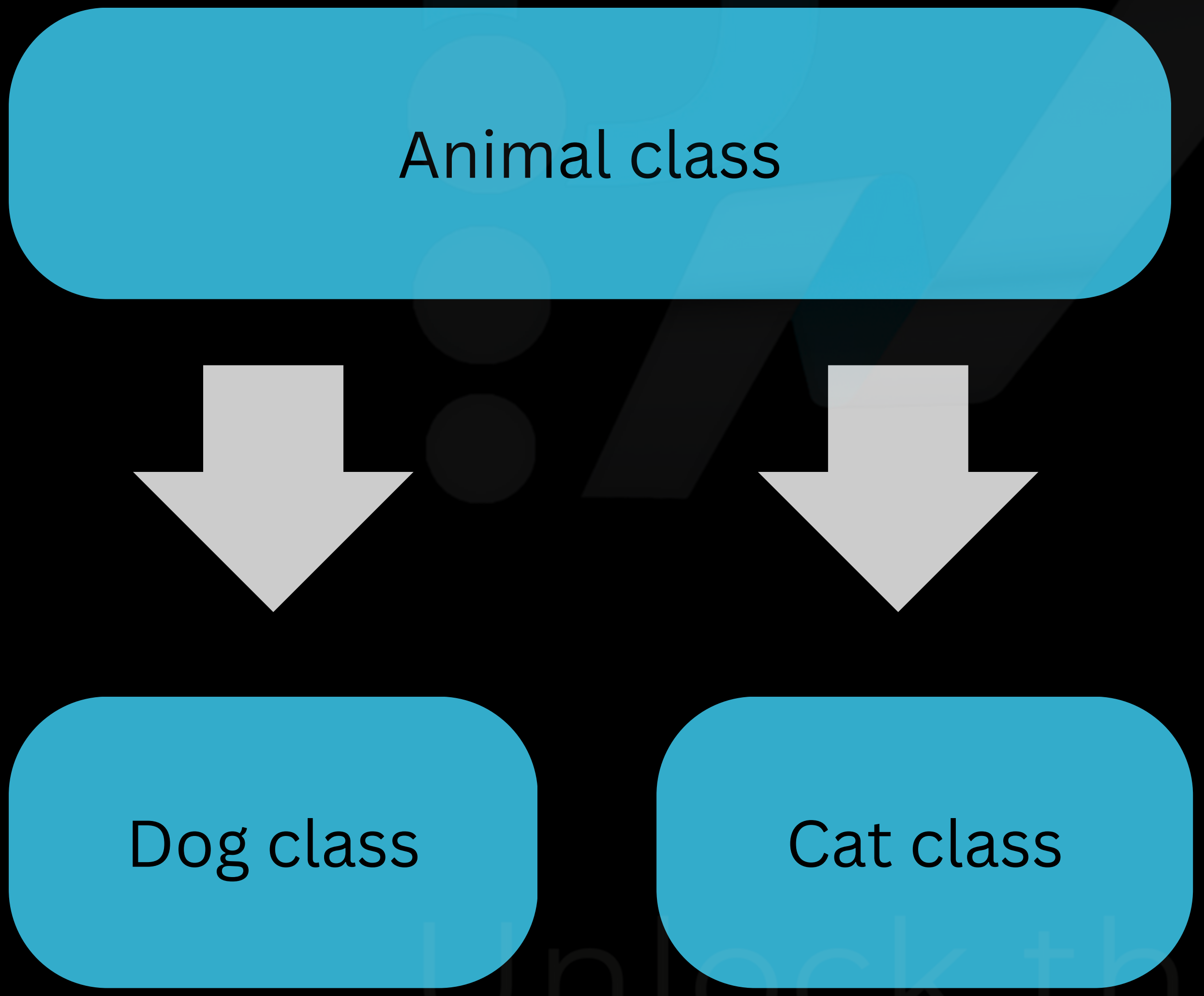


Example

```python
class Grandparent:
    def grandparent_method(self):
        return "Grandparent's method"

class Parent(Grandparent):
    def parent_method(self):
        return "Parent's method"

class Child(Parent):
    def child_method(self):
        return "Child's method"


# Usage
child = Child()
print(child.grandparent_method())
# Outputs: Grandparent's method
print(child.parent_method())
# Outputs: Parent's method
```

Hierarchical Inheritance

In hierarchical inheritance, multiple child classes inherit from a single parent class.

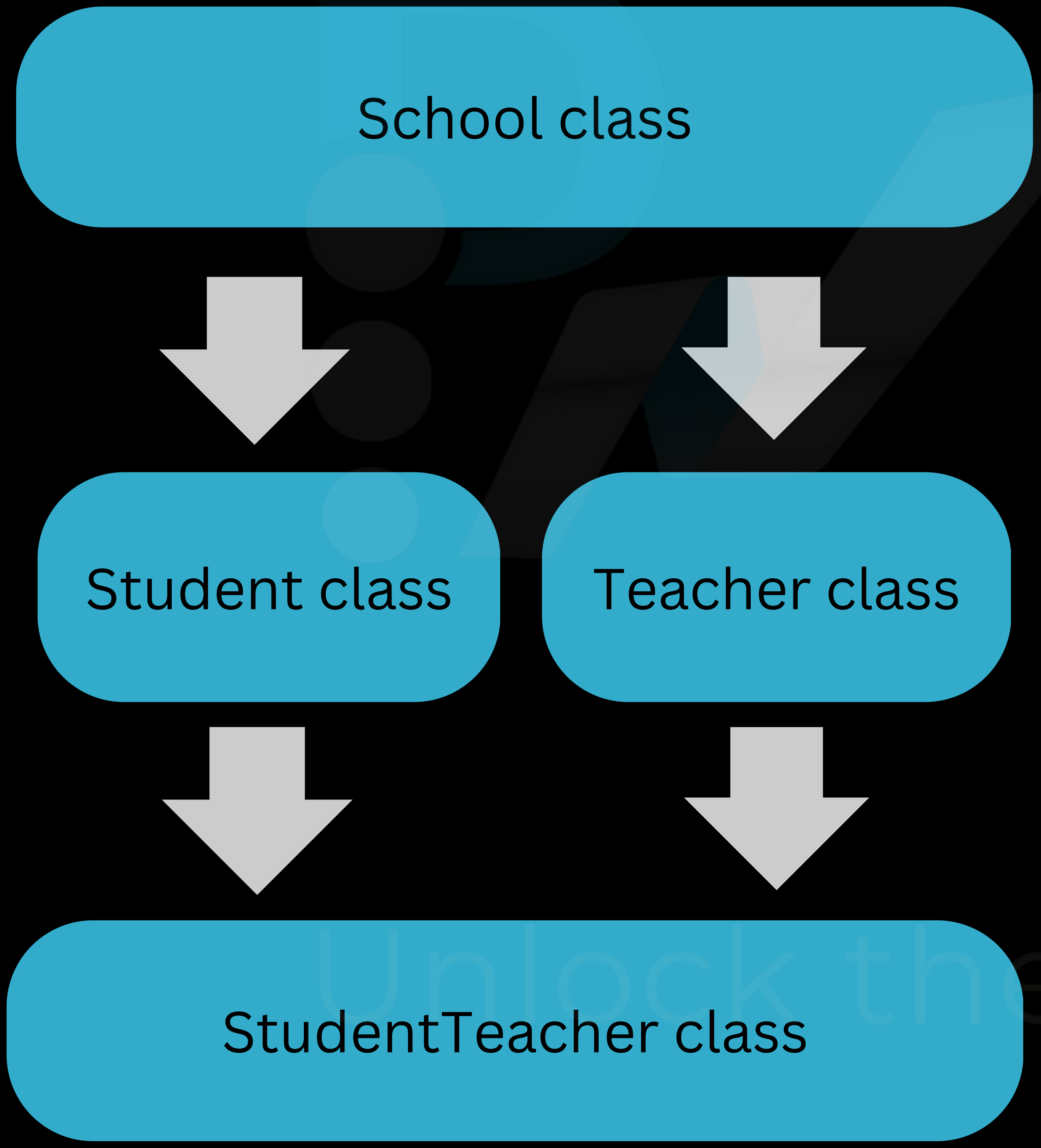


Example

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

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

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

# Usage
dog = Dog()
cat = Cat()
print(dog.speak())  # Outputs: Dog barks
print(cat.speak())  # Outputs: Cat meows
```

Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance.



Example

```
class School:
    def school_name(self):
        return "ABC School"

class Student(School):
    def student_info(self):
        return "Student class"

class Teacher(School):
    def teacher_info(self):
        return "Teacher class"

class StudentTeacher(Student, Teacher):
    def student_teacher_info(self):
        return "Student Teacher class"

# Usage
st = StudentTeacher()
print(st.school_name())     # Outputs: ABC School
print(st.student_info())    # Outputs: Student
class
print(st.teacher_info())    # Outputs: Teacher class
```

Data Abstraction

Data abstraction is a concept in object-oriented programming that hides unnecessary details from the user and only shows the essential features of an object. It helps in reducing complexity and increasing code readability.

How Does Abstraction Work ?

- In Python, abstraction is achieved using abstract classes and abstract methods.
- An abstract class is a class that cannot be instantiated (you cannot create an object of it).
- It contains abstract methods (methods without implementation) that must be implemented in the child class.

Think of data abstraction like a TV remote control. You just need to know which buttons to press, but you don't need to know how it works inside!

Example using a Mobile Phone

Think of it this way :

- When you use your real mobile phone, you just press the power button
- You don't need to know how the battery works inside
- You just need to know how to check battery level

This is exactly what abstraction does :

1. Hides complicated stuff inside (using __)
2. Gives you simple methods to use (like switch_on())
3. Protects the data from accidental changes
4. Makes the code easier to use

READ MORE

Example

```
class MobilePhone:
    def __init__(self):
        self.__battery_level = 100
        self.__is_on = False

    def switch_on(self):
        self.__is_on = True
        print("Phone is switched ON")

    def switch_off(self):
        self.__is_on = False
        print("Phone is switched OFF")

    def check_battery(self):
        return f"Battery level: {self.__battery_level}%"

# Using the phone
my_phone = MobilePhone()
my_phone.switch_on()
# Output: Phone is switched ON
print(my_phone.check_battery())
# Output: Battery level: 100%
```

Polymorphism

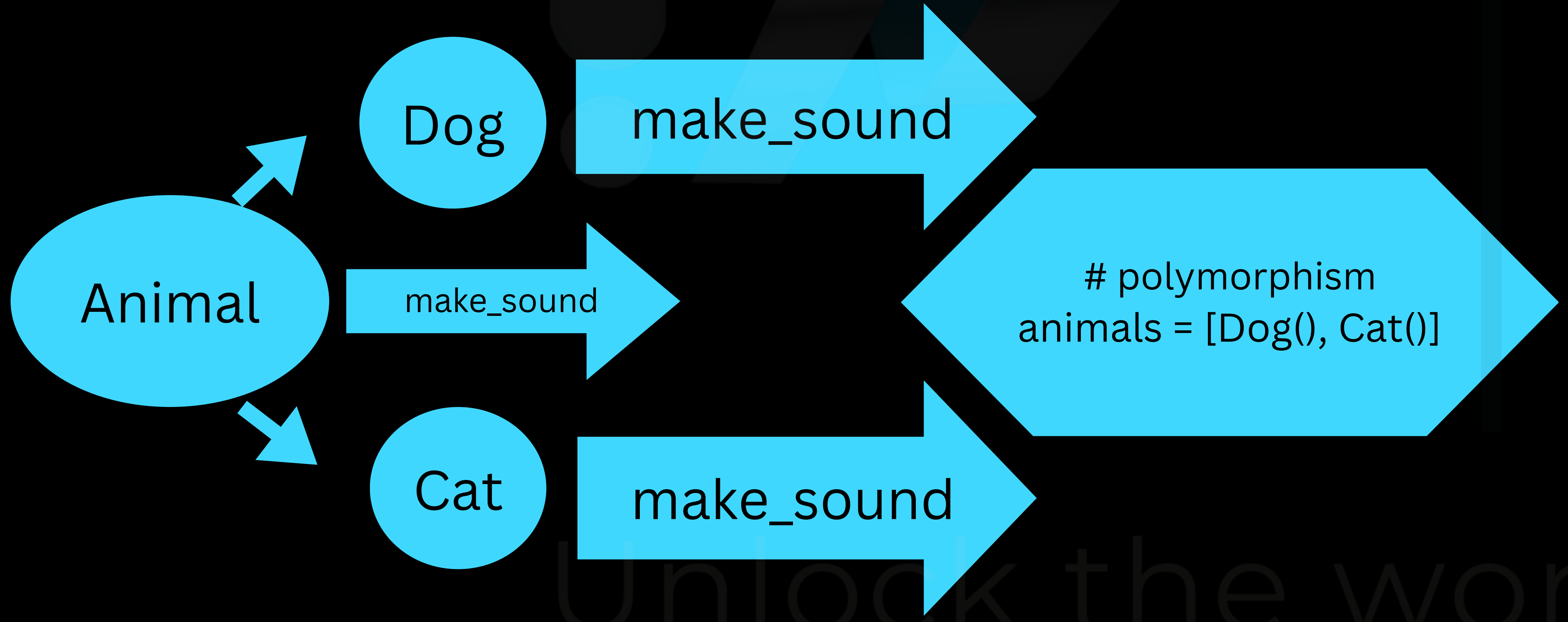
Polymorphism means "many forms" in Greek. In Python, polymorphism allows objects of different classes to be treated as objects of a common class. It helps in writing flexible and reusable code.

Types of Polymorphism in Python

1. Method Overriding (Runtime Polymorphism)
2. Method Overloading (Python does not support true method overloading but can be achieved using default arguments)
3. Operator Overloading

Method Overriding (Runtime Polymorphism)

When a child class provides a specific implementation of a method that is already defined in its parent class.



Example

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

class Dog(Animal):
    def make_sound(self):
        # Overriding parent method
        print("Dog barks")

class Cat(Animal):
    def make_sound(self):
        # Overriding parent method
        print("Cat meows")

# Using polymorphism
animals = [Dog(), Cat()]
for animal in animals:
    animal.make_sound()

# Output:
# Dog barks
# Cat meows
```

Method Overloading

(Not Directly Supported in Python)

Python does not support method overloading like Java/C++, but it can be done using default arguments.

Example

```
class MathOperations:
    def add(self, a, b, c=0):
        # Default argument c
        return a + b + c

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

Operator Overloading

Python allows operators like +, -, * to work differently for different data types by defining special methods like __add__(), __sub__(), etc.

Example

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

    def __add__(self, other):
        # Overloading '+' operator
        return Number(self.value +
other.value)

num1 = Number(5)
num2 = Number(10)
result = num1 + num2
 # Calls __add__() method
print(result.value)
# Output: 15
```

✅ **Polymorphism**

allows the same method name to have different behaviors.

✅ **Method overriding**

lets child classes redefine a parent class method.

✅ **Method overloading**

can be simulated using default arguments.

✅ **Operator overloading**

lets us use operators with custom classes.

READ MORE