Data Abstraction in Python

Juice Vending Machine

Apple Juice

- You don’t see or do the inside work.
- You just get your juice!

This is abstraction
— you only deal with what you need (buttons + juice), the rest is hidden.

Same Idea in Python

When we write Python programs, we do the same thing

- We create classes and methods that hide the inside steps.
- The user only uses the final part

— like pressing a button.

```python
class JuiceMachine:
    def make_apple_juice(self):
        # hidden steps inside
        self.__wash_apples()
        self.__cut_apples()
        self.__squeeze_apples()
        print("Here is your apple juice!")

    def __wash_apples(self):
        print("Washing apples...")

    def __cut_apples(self):
        print("Cutting apples...")

    def __squeeze_apples(self):
        print("Squeezing apples...")

# User just presses button
machine = JuiceMachine() machine.make_apple_juice()
```

Is like the button on the machine.

The user doesn't see or worry about these hidden steps.

Washing
cutting
squeezing

Hidden
(marked with _ to make them private)

Key Idea

- You get your juice → you don’t care how.
- In Python → You use a method → you don’t care how it works inside.
- This is abstraction!

Technical Definition

Data Abstraction is a principle of object-oriented programming (OOP) that hides internal implementation details and shows only the relevant features of an object or class to the user.

In Python, abstraction is typically implemented using

- Abstract classes (created using the ABC module).
- Abstract methods (methods that are declared but contain no implementation in the base class).

Subclasses inherit the abstract class and provide concrete implementations for the abstract methods.

- **Purpose:** Reduce complexity by hiding unnecessary details.
- **How:** Using ABC (Abstract Base Class) and @abstractmethod decorators.
- **Benefit:** Provides a clear and simple interface while hiding complex logic.

```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 * self.radius

# Cannot create Shape() directly because it's abstract
circle = Circle(5)
print(circle.area())
```

Is an abstract method
→ must be defined in the child class.

Is an abstract class
→ can't be used directly.

is a concrete class
→ provides the actual implementation.

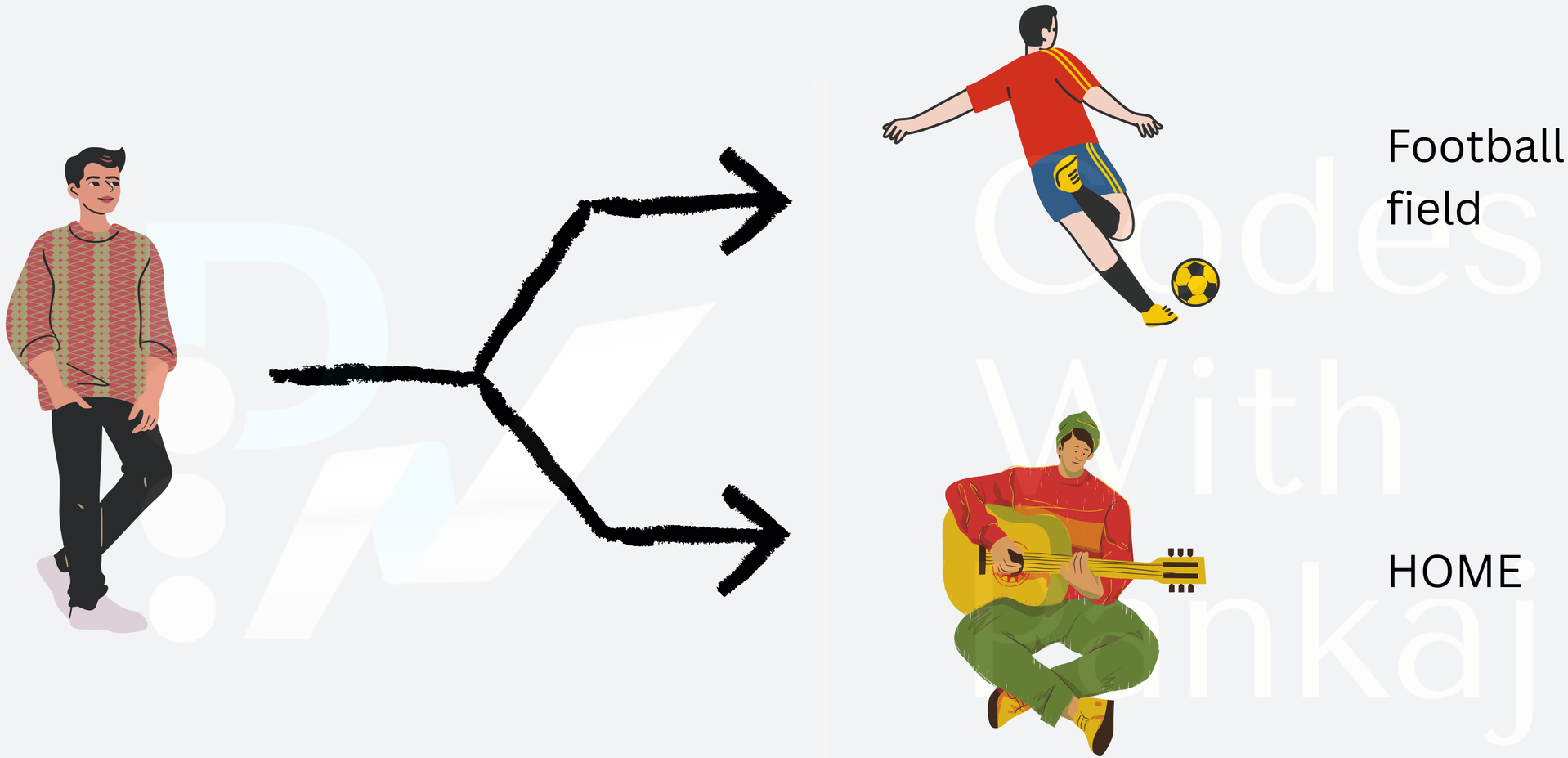
Data abstraction lets programmers

- Create blueprints for objects (abstract classes).
- Force child classes to follow rules (implement abstract methods).
- Hide the complex code — only show what the user needs (interface).

Polymorphism in Python

What is Polymorphism?

{ “Same name, different work.” }



Imagine :

- You have a friend named Alex.
- Alex can be a student in class, a player on the football field, and a musician at home.

It’s still the same Alex, but Alex behaves differently depending on where he is and what he’s doing.

In Python, this happens in two main ways

Built-in polymorphism
(like the print() example)

Polymorphism in classes
(using methods)

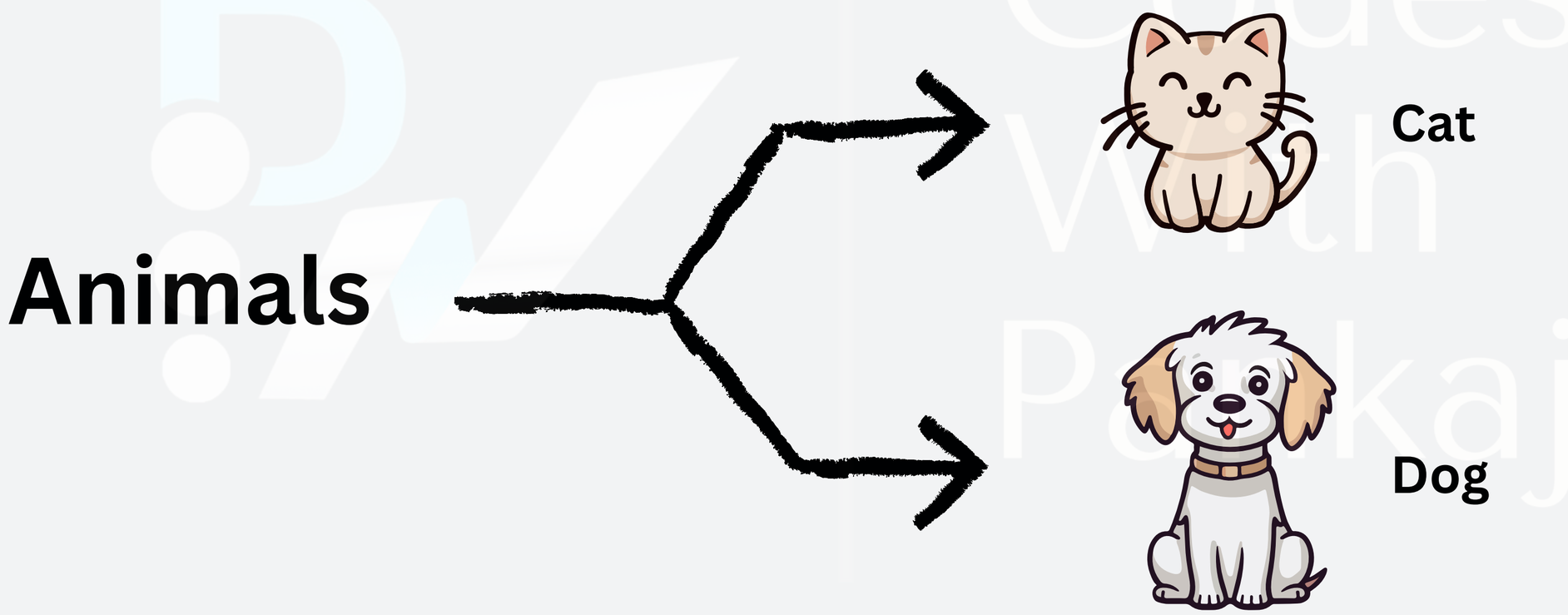
Built-in Polymorphism

Python functions like len(), + operator, or print() work differently with different data types.

```
print(len("Hello"))    # Counts letters in string → 5
print(len([1, 2, 3])) # Counts items in list → 3
```

Same function → Works differently depending on what you give it.

Polymorphism with Classes



Both are animals, but they make different sounds.

Without Polymorphism

```
class Dog:
    def bark(self):
        print("Woof!")

class Cat:
    def meow(self):
        print("Meow!")
```

- To make them speak, you must remember different method names (bark(), meow()).

With Polymorphism

Let’s use the same method name: speak()

```
1  class Dog:
2      def speak(self):
3          print("Woof!")
4
5  class Cat:
6      def speak(self):
7          print("Meow!")
8  # --------------------
9  dog = Dog()
10 cat = Cat()
11
12 for animal in (dog, cat):
13     animal.speak()
14
```

speak() is the same name
– but each animal uses it in its own way

Same name: speak()

Different forms :
Dog says Woof, Cat says Meow

Another Example: Polymorphism with Functions

Suppose you make a function that works for any shape

```python
class Square:
    def area(self):
        return 4 * 4   # Example

class Circle:
    def area(self):
        return 3.14 * 2 * 2   # Example

def print_area(shape):
    print(shape.area())

s = Square()
c = Circle()

print_area(s)   # 16
print_area(c)   # 12.56
```

function works for any shape

— it doesn't care if it's a square or a circle

— because both have an area() method.

Again, same name, different behavior = Polymorphism!