**James Whitcomb Riley** said

"When I see a bird that **walks** like a duck, **swims** like a duck, and **quacks** like a duck,
I call that bird a duck."

This quote beautifully captures the **essence of Duck Typing** in programming.

# Polymorphism

Python is **polymorphic**

# Polymorphism:

- **Polymorphism** *means* "**many forms**"
- In programming, this means:
- The same **function** or **operation** behaves **differently** depending on the **object it's acting on**.
- It allows the same interface or method to behave differently based on the **object**
- It is a concept in object-oriented programming (OOP) that allows **objects** of **different classes** to be treated the **same**
- It shifts the focus from **data types** to **behaviours**

# Python-style Polymorphism

- Python is **dynamically typed**, so it handles polymorphism pretty gracefully.

There are **two** main ways to think about it in Python:

1. **Duck Typing (Informal Polymorphism)**

**“If it walks like a duck and quacks like a duck, it's a duck.”**

Python doesn’t care what the actual type is — it just checks if the object can do what you’re asking it to do.

# Duck-Typing

- **Duck Typing** is a type system used in **dynamic languages**
- For example, **Python**, **Perl**, **Ruby**, **PHP**, **Javascript**, etc. where the type or the **class** of an **object** is **less important** than the **method it defines**
- Using **Duck Typing**, we do not check **types** at all
- Instead, we check for the presence of a given method or attribute

# Example:

```
class Dog:
    def speak(self):
        return "Waoo Waoo"

class Cat:
    def speak(self):
        return "Myaaon Myaaon!"

def animal_sound(animal):
    print(animal.speak())

animal_sound(Dog())  # Woof!
animal_sound(Cat())  # Meow!
```

**So in Python**, instead of checking **is this an Animal?**

we just ask: **“Can it speak()?”**

# 2. Inheritance + Method Overriding

(Formal Polymorphism)

- When a **base class** defines a method, and **derived classes override** it with their **own** behaviour

**Inheritance** sets the foundation for **method overriding**, which is key to achieving **polymorphism**

# Example:

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

class Dog(Animal):
    def speak(self):
        return "Kochek Raashagaa en, wakkaga en"

class Cat(Animal):
    def speak(self):
        return "Myaaooon...."

animals = [Dog(), Cat(), Animal()]
for animal in animals:
    print(animal.speak())
```

# Let’s discuss some more on Polymorphism…

- **Function Polymorphism** (**Built-in Polymorphism**)

Some Python **built-in functions** automatically work with different data types — that's polymorphism too

How and why?

# Examples:

```
print(len("Balochistan"))    # 11 (number of characters)
print(len([1, 2, 3]))        # 3 (number of elements in list)
print(len((10, 20, 30, 40)))# 4 (number of elements in tuple)

print(max(5, 9, 2))                # 9
print(max("a", "z", "b"))          # 'z' (based on ASCII value)
print(max(["apple", "banana"]))  # 'banana' (alphabetically last)

print(sum([1, 2, 3]))         # 6
print(sum((10.5, 20.5)))     # 31.0

print(type("Hello"))          # <class 'str'>
print(type(100))              # <class 'int'>
print(type([1, 2, 3]))         # <class 'list'>
```

# Why this is Polymorphism?

- These functions don’t care about the type of object--they adapt their behaviour depending on it
- You use the same function name (len, max, etc.), but it acts differently based on the input type.
- That’s the beauty of **polymorphism**!

# Some Questions For you…

- What is Method Overloading and does it work with Python?
- What is Operator Overloading and How is this linked with Polymorphism?
- How do you relate Encapsulation with Polymorphism? How do they work together?

**Search for the above questions and answer with examples…**