Polymorphism in Python: Simple, Complex & Industry-level Examples

# 1. Introduction

Polymorphism allows for the same interface to be used for different underlying forms (data types). In object-oriented programming, polymorphism refers to the ability to redefine methods in derived classes. Python supports multiple types of polymorphism, including duck typing, operator overloading, method overriding, and interface implementation through abstract base classes.

# 2. Simple Example: Duck Typing

This shows how different objects can implement the same method name, and Python calls the appropriate method at runtime.

Example:  
  
class Bird:  
 def sound(self):  
 return "Tweet"  
  
class Dog:  
 def sound(self):  
 return "Bark"  
  
animals = [Bird(), Dog()]  
for animal in animals:  
 print(animal.sound())

# 3. Medium Example: Operator Overloading

Operator overloading allows us to redefine how Python operators behave for user-defined classes.

Example:  
  
class Vector:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def \_\_add\_\_(self, other):  
 return Vector(self.x + other.x, self.y + other.y)  
  
 def \_\_str\_\_(self):  
 return f"Vector({self.x}, {self.y})"  
  
v1 = Vector(2, 3)  
v2 = Vector(4, 5)  
print(v1 + v2) # Vector(6, 8)

# 4. Complex Example: Abstract Base Class & Method Overriding

This approach is often used in frameworks and large-scale applications where you want to enforce a contract.

Example:  
  
from abc import ABC, abstractmethod  
  
class PaymentProcessor(ABC):  
 @abstractmethod  
 def pay(self, amount):  
 pass  
  
class StripeProcessor(PaymentProcessor):  
 def pay(self, amount):  
 print(f"Processing ${amount} through Stripe")  
  
class PayPalProcessor(PaymentProcessor):  
 def pay(self, amount):  
 print(f"Processing ${amount} through PayPal")  
  
def process\_payment(processor: PaymentProcessor, amount):  
 processor.pay(amount)  
  
process\_payment(StripeProcessor(), 100)  
process\_payment(PayPalProcessor(), 200)

# 5. Industry-Level Example: Notification Service

Imagine a system that sends notifications via Email, SMS, or Push. Each type implements the same interface.

Example:  
  
from abc import ABC, abstractmethod  
  
class Notifier(ABC):  
 @abstractmethod  
 def notify(self, message):  
 pass  
  
class EmailNotifier(Notifier):  
 def notify(self, message):  
 print(f"Email: {message}")  
  
class SMSNotifier(Notifier):  
 def notify(self, message):  
 print(f"SMS: {message}")  
  
class PushNotifier(Notifier):  
 def notify(self, message):  
 print(f"Push: {message}")  
  
def notify\_users(notifiers, message):  
 for notifier in notifiers:  
 notifier.notify(message)  
  
notifiers = [EmailNotifier(), SMSNotifier(), PushNotifier()]  
notify\_users(notifiers, "Service outage at 3PM")

# 6. Conclusion

Polymorphism is a powerful concept for writing clean, flexible, and maintainable Python code. Whether it's duck typing for quick flexibility or abstract base classes for large-scale systems, polymorphism scales from small to enterprise-level applications.