Table of Contents

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. The word "polymorphism" comes from Greek meaning "many forms".

1. What is Polymorphism?

Polymorphism lets you:

- Use the same method name across different classes
- Have different implementations of the same method
- Call methods without knowing the exact class of the object

2. Types of Polymorphism in Python

1. **Method Overriding**: When a child class provides a different implementation of a method that is already defined in its parent class.
2. **Method Overloading**: Python doesn't support this directly like Java/C++, but we can achieve similar functionality using default arguments or variable-length arguments.
3. **Duck Typing**: "If it walks like a duck and quacks like a duck, it must be a duck" - we don't check types, just behavior.

3. Examples

1. Method Overriding Example

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):  # Overriding the speak method
        print("Dog barks")

class Cat(Animal):
    def speak(self):  # Overriding the speak method
        print("Cat meows")

# Polymorphic behavior
animals = [Animal(), Dog(), Cat()]
```

```
for animal in animals:
    animal.speak()
```

Output:

```
Animal speaks
Dog barks
Cat meows
```

2. Duck Typing Example

```
class Parrot:
    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:
    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")

# Common interface
def flying_test(bird):
    bird.fly()

# Instantiate objects
parrot = Parrot()
penguin = Penguin()

# Polymorphic behavior
flying_test(parrot)
flying_test(penguin)
```

Output:

```
Parrot can fly
Penguin can't fly
```

3. Operator Overloading Example

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Overloading the + operator
    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(1, 2)
print(v1 + v2)  # Uses the overloaded + operator
```

Output:

```
Vector(3, 5)
```

4. Real-World Example: Payment Processing System

```
class Payment:
    def process_payment(self, amount):
        pass

class CreditCardPayment(Payment):
    def process_payment(self, amount):
        print(f"Processing credit card payment of ${amount}")

class PayPalPayment(Payment):
    def process_payment(self, amount):
        print(f"Processing PayPal payment of ${amount}")

class BankTransferPayment(Payment):
    def process_payment(self, amount):
        print(f"Processing bank transfer of ${amount}")

def checkout(payment_method, amount):
    payment_method.process_payment(amount)

# Usage
credit_card = CreditCardPayment()
paypal = PayPalPayment()
bank_transfer = BankTransferPayment()

checkout(credit_card, 100)    # Processing credit card payment of $100
```

```
checkout(paypal, 50)          # Processing PayPal payment of $50
checkout(bank_transfer, 200)  # Processing bank transfer of $200
```

In this example, the `checkout` function doesn't need to know what specific type of payment method it's using - it just calls `process_payment()`. Each payment type implements this method differently, demonstrating polymorphism.

Key Benefits of Polymorphism

1. **Code reusability**: Write more generic and reusable code
2. **Flexibility**: Easily extend and modify code
3. **Simplified syntax**: Work with objects at a higher level of abstraction
4. **Maintainability**: Easier to maintain and update code

Polymorphism is a powerful concept that helps make your Python code more flexible and easier to work with as your programs grow in complexity.

---

5. Tasks

Task 1: Animal Sounds

Objective: Create classes with method overriding.
Problem:

- Create a base class `Animal` with a `make_sound()` method that prints "Generic animal sound".
- Create 3 subclasses (e.g., `Dog`, `Cat`, `Cow`) that override `make_sound()` with their own sounds.
- Store instances of all classes in a list and call `make_sound()` for each.

Example Output:

```
Generic animal sound
Woof!
Meow!
Moo!
```

Task 2: Shape Area Calculator

Objective: Use polymorphism to calculate areas.
Problem:

- Create a base class `Shape` with an abstract `area()` method (use `pass`).
- Create subclasses `Circle`, `Square`, and `Triangle` that implement `area()`.
- Use a loop to calculate and print areas of different shapes.

Requirements:

- `Circle`: Initialize with radius, area = πr^2

- `Square`: Initialize with side length, area = side²
- `Triangle`: Initialize with base/height, area = 0.5 * base * height

---

Task 3: Duck Typing in Action

Objective: Practice "duck typing" with unrelated classes.
Problem:

- Create two unrelated classes: `Car` and `Bird`.
- Both classes should have a `move()` method but with different implementations:
  - `Car.move()` prints "Car is driving"
  - `Bird.move()` prints "Bird is flying"
- Create a function `travel(object)` that calls `object.move()`.
- Demonstrate polymorphism by passing both objects to `travel()`.

**Task 4: Operator Overloading (+ and)*

Objective: Overload operators for a custom class.
Problem:

- Create a `Book` class with attributes `title` and `pages`.
- Overload the `+` operator to combine two books into a new "collection" book:
  - New title = "Collection: [Book1 Title] & [Book2 Title]"
  - New pages = sum of both books' pages
- Example:

```
book1 = Book("Python Basics", 100)
book2 = Book("OOP Guide", 150)
book3 = book1 + book2
print(book3.title)  # "Collection: Python Basics & OOP Guide"
print(book3.pages)  # 250
```

Task 5: Media Player System

Objective: Simulate a real-world polymorphic system.
Problem:

- Create a base class `MediaPlayer` with a `play()` method.
- Create subclasses for different media types: `MP3Player`, `VideoPlayer`, `StreamingPlayer`.
- Each subclass should override `play()` with specific behavior:
  - `MP3Player`: "Playing audio track..."
  - `VideoPlayer`: "Playing video..."
  - `StreamingPlayer`: "Streaming from cloud..."
- Create a list of different players and call `play()` for each.

Task 6: Bank Account Interest

Objective: Polymorphism in financial calculations.
Problem:

- Create a base class `BankAccount` with a `calculate_interest()` method.
- Create subclasses `SavingsAccount` (5% interest) and `FixedDeposit` (7% interest).
- Initialize accounts with a balance, then calculate and print interest for different account types.

Example:

```
savings = SavingsAccount(1000)
fixed = FixedDeposit(1000)
print(savings.calculate_interest())  # 50.0
print(fixed.calculate_interest())    # 70.0
```

Task 7: Math Operations

Objective: Use polymorphism with numbers.
Problem:

- Create a function `double(x)` that returns `x * 2`.
- Show that it works with integers, floats, lists, and strings (demonstrating Python's built-in polymorphism).
- Example outputs:

```
print(double(5))     # 10
print(double(3.14))  # 6.28
print(double([1,2])) # [1,2,1,2]
print(double("Hi"))  # "HiHi"
```

Task 8: Game Characters

Objective: Polymorphic behavior in a game.
Problem:

- Create a base class `GameCharacter` with an `attack()` method.
- Create subclasses `Warrior`, `Mage`, and `Archer` with unique attack messages:
  - Warrior: "Sword slash!"
  - Mage: "Fireball!"
  - Archer: "Arrow shot!"
- Create a list of characters and call `attack()` for each.

Task 9: File Handling

Objective: Polymorphic file operations.
Problem:

- Create classes `TextFile` and `CSVFile`, both with a `read()` method.
- `TextFile.read()` returns the text file's content as a string.
- `CSVFile.read()` returns the data as a list of lists.
- Write a function `process_file(file)` that calls `read()` on any file type.

Task 10: Advanced Challenge - Logger System

Objective: Design a flexible logging system.
Problem:

- Create a base class `Logger` with a `log(message)` method.
- Implement 3 subclasses: `ConsoleLogger`, `FileLogger`, and `DatabaseLogger`.
- Each subclass should log messages differently:
  - Console: Print to screen
  - File: Append to a text file
  - Database: Simulate saving to a DB (print "Saved to DB: [message]")
- Create a list of loggers and test all logging methods.

Tips for Success:

1. Start with simple tasks like **Task 1** or **Task 5**.
2. Use method overriding for most tasks.
3. For duck typing (**Task 3**), focus on shared method names rather than inheritance.
4. Test your code with different object types.

Related Topics

- **Python OOP Basics** – Learn how to create and use classes in Python with practical examples. Master constructors, methods, attributes, and OOP principles for efficient coding.
  - 👉 Learn more
- **Python Inheritance** - Learn Python OOP inheritance with Beginner's examples! Understand parent & child classes, method overriding, `super()`, and multilevel inheritance.
  - 👉 Learn more