Polymorphism in Python

What Is Polymorphism?

Polymorphism means:

> **One interface, different behavior.**

The word comes from:

- **Poly** = many
- **Morph** = forms

In programming, polymorphism allows different objects to respond to the **same method or operation in their own way**.

For example, imagine different types of resources on haas.dev:

- A PDF resource can be opened.
- A video resource can be played.
- A quiz resource can be started.

They are different objects, but each can provide its own implementation of a common action.

```
class PDF:
    def open(self):
        print("Opening PDF")


class Video:
    def open(self):
        print("Playing video")
```

```
class Quiz:
    def open(self):
        print("Starting quiz")
```

Now:

```
resources = [PDF(), Video(), Quiz()]

for resource in resources:
    resource.open()
```

Output:

```
Opening PDF
Playing video
Starting quiz
```

The loop does not need to know which type of object it is working with.

It simply says:

```
resource.open()
```

Each object decides what `open()` means for itself.

Why Does Polymorphism Matter?

Without polymorphism, code often becomes filled with type checks.

For example:

```
for resource in resources:
    if isinstance(resource, PDF):
        print("Opening PDF")
    elif isinstance(resource, Video):
        print("Playing video")
    elif isinstance(resource, Quiz):
        print("Starting quiz")
```

This works, but it becomes difficult to maintain.

If you add another resource:

```
class Article:
    ...
```

you may need to modify the loop again.

With polymorphism:

```
for resource in resources:
    resource.open()
```

The loop stays the same.

This gives you an important design principle:

> **Code should depend on what an object can do, not unnecessarily on what type it is.**

A Simple Real World Example

Think about a payment system.

Different payment methods may have:

```
pay()
```

But the behavior is different.

```
Credit Card  processes card payment
PayPal       processes PayPal payment
Bank         processes bank transfer
```

The application can simply call:

```
payment.pay()
```

without needing to know the internal details of each payment method.

Polymorphism Through Inheritance

One common way to achieve polymorphism in Python is through **inheritance**.

Suppose we have a parent class:

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

Now create child classes:

```
class Dog(Animal):
    def speak(self):
        print("Dog says: Woof")


class Cat(Animal):
    def speak(self):
        print("Cat says: Meow")
```

Now:

```
dog = Dog()
cat = Cat()

dog.speak()
cat.speak()
```

Output:

```
Dog says: Woof
Cat says: Meow
```

Both objects have the same method:

```
speak()
```

But each provides different behavior.

That is polymorphism.

Method Overriding and Polymorphism

Polymorphism often works together with **method overriding**.

Method overriding happens when a child class provides its own implementation of a method inherited from its parent.

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


class Dog(Animal):
    def speak(self):
        print("Woof")


class Cat(Animal):
    def speak(self):
        print("Meow")
```

Here:

`Dog.speak()`

and:

`Cat.speak()`

override the parent implementation.

Then:

```
animals = [Dog(), Cat(), Animal()]

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

Output:

```
Woof
Meow
Some sound
```

The same call:

```
animal.speak()
```

produces different results depending on the object.

The Important Idea: Same Interface

Polymorphism is not simply:

> "Different classes have different methods."

The important part is that different objects can be used through a **common interface**.

For example:

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

Every object provides:

`speak()`

The calling code does not need to know the exact implementation.

This separates:

What the object can do

from:

How the object does it.

Polymorphism Without Inheritance

Python does not require inheritance for polymorphism.

This is one of the most important things to understand about Python.

Consider:

```
class Dog:
    def speak(self):
        print("Woof")


class Robot:
    def speak(self):
        print("Beep")
```

```
class Person:
    def speak(self):
        print("Hello")
```

These classes have no parent-child relationship.

But:

```
things = [Dog(), Robot(), Person()]

for thing in things:
    thing.speak()
```

Output:

```
Woof
Beep
Hello
```

Python only cares that the objects provide:

```
speak()
```

This is commonly associated with **duck typing**.

Duck Typing

Python follows a principle often described as:

>

> "If it behaves like the required object, it can be used."

The idea comes from the phrase:

> "If it walks like a duck and quacks like a duck, treat it like a duck."

Suppose a function needs an object that can `open()`:

```
def open_resource(resource):
    resource.open()
```

We can pass different objects:

```
class PDF:
    def open(self):
        print("Opening PDF")


class Website:
    def open(self):
        print("Opening website")


class Video:
    def open(self):
        print("Playing video")
```

Now:

```
open_resource(PDF())
```

```
open_resource(Website())
open_resource(Video())
```

Output:

```
Opening PDF
Opening website
Playing video
```

The function does not check:

```
isinstance(resource, PDF)
```

It simply expects the object to provide:

```
open()
```

Why Duck Typing Is Useful

Imagine building a file processing system.

You might have:

```
class PDFFile:
    def process(self):
        print("Processing PDF")


class CSVFile:
    def process(self):
        print("Processing CSV")
```

```
class JSONFile:
    def process(self):
        print("Processing JSON")
```

Then:

```
def process_file(file):
    file.process()
```

You can use:

```
process_file(PDFFile())
process_file(CSVFile())
process_file(JSONFile())
```

The function doesn't need separate logic for every file type.

This makes code easier to extend.

Polymorphism With Functions

Polymorphism can be used with ordinary functions as well.

```
class Circle:
    def area(self):
        return 50


class Rectangle:
```

```
    def area(self):
        return 100


def show_area(shape):
    print(shape.area())
```

Now:

```
show_area(Circle())
show_area(Rectangle())
```

Output:

```
50
100
```

The function works with different objects because they provide the same behavior.

Polymorphism With Built In Functions

Python itself uses polymorphism extensively.

Consider:

```
print(len("Python"))
print(len([1, 2, 3]))
print(len((10, 20)))
```

Output:

```
6
3
2
```

The same function:

```
len()
```

works with different types.

It knows how to determine the length of:

- strings
- lists
- tuples
- sets
- dictionaries
- many other objects

The exact internal implementation depends on the object.

This is polymorphic behavior.

Operator Polymorphism

Operators can also behave differently depending on the data type.

For example:

```
print(2 + 3)
```

Output:

```
5
```

But:

```
print("Hello " + "World")
```

Output:

```
Hello World
```

The `+` operator is being used with different types.

For numbers:

```
2 + 3
```

means numerical addition.

For strings:

```
"Hello " + "World"
```

means concatenation.

The same operator provides different behavior depending on the operands.

Polymorphism With Custom Classes

Python allows you to define how operators behave for your own classes.

For example:

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

    def __add__(self, other):
        return Point(
            self.x + other.x,
            self.y + other.y
        )
```

Now:

```
p1 = Point(2, 3)
p2 = Point(4, 5)

p3 = p1 + p2

print(p3.x)
print(p3.y)
```

Output:

```
6
8
```

The `+` operator now has meaning for `Point` objects.

This is closely related to **dunder methods**, which will be covered separately.

A Practical Example: Notifications

Imagine an application that supports multiple notification systems.

```
class EmailNotification:
    def send(self, message):
        print(f"Email: {message}")


class SMSNotification:
    def send(self, message):
        print(f"SMS: {message}")


class PushNotification:
    def send(self, message):
        print(f"Push: {message}")
```

We can create:

```
notifications = [
    EmailNotification(),
    SMSNotification(),
    PushNotification()
]
```

Then:

```
for notification in notifications:
    notification.send("Your order is ready")
```

Output:

```
Email: Your order is ready
SMS: Your order is ready
Push: Your order is ready
```

The application only needs to know:

```
notification.send(...)
```

It does not need separate logic for each notification type.

A haas.dev Example

Imagine haas.dev has different resource types.

```
class Resource:
    def open(self):
        raise NotImplementedError
```

Now:

```
class PDFResource(Resource):
    def open(self):
        print("Opening PDF guide")


class QuizResource(Resource):
    def open(self):
        print("Starting quiz")


class VideoResource(Resource):
    def open(self):
```

```
        print("Playing video")
```

We can create:

```
resources = [
    PDFResource(),
    QuizResource(),
    VideoResource()
]
```

Then:

```
for resource in resources:
    resource.open()
```

Output:

```
Opening PDF guide
Starting quiz
Playing video
```

The resource manager does not need to know exactly which resource it received.

It only needs to know:

```
resource.open()
```

This is a practical example of polymorphism making a system easier to extend.

Adding New Types

Suppose haas.dev later adds an interactive coding exercise.

We can create:

```
class CodingExercise(Resource):
    def open(self):
        print("Starting coding exercise")
```

The existing loop does not need to change:

```
for resource in resources:
    resource.open()
```

This is one of the biggest advantages of polymorphic design.

You can add new behavior without constantly rewriting existing code.

Polymorphism and Abstraction

Polymorphism and abstraction are related, but they are not the same thing.

Abstraction

Focuses on:

> **What should an object provide?**

For example:

`open()`

Polymorphism

Focuses on:

> **How can different objects provide that behavior differently?**

For example:

```
PDF    → open PDF
Video  → play video
Quiz   → start quiz
```

A simple way to remember:

```
Abstraction → common contract
Polymorphism → different implementations
```

Polymorphism and Inheritance

Inheritance answers:

> "Is this class a specialized version of another class?"

Example:

`Resource`

PDFResource

Polymorphism answers:

> "Can different objects be used through the same interface?"

Example:

```
PDFResource
QuizResource
VideoResource
    
  open()
```

Inheritance can enable polymorphism, but Python polymorphism does **not** require inheritance.

Common Mistake: Checking Types Everywhere

Avoid code like:

```
def process(resource):
  if isinstance(resource, PDF):
    ...
  elif isinstance(resource, Video):
    ...
  elif isinstance(resource, Quiz):
    ...
```

Sometimes type checking is appropriate, but if every new class requires another `if` or `elif`, your design may benefit from polymorphism.

Instead:

```
def process(resource):
    resource.open()
```

Each object handles its own behavior.

Common Mistake: Different Method Names

Suppose you have:

```
class PDF:
    def open(self):
        print("Opening PDF")


class Video:
    def play(self):
        print("Playing video")
```

Now this becomes difficult:

```
for resource in resources:
    # Which method should be called?
```

A common interface makes the design easier:

```
class PDF:
    def open(self):
        print("Opening PDF")
```

```
class Video:
    def open(self):
        print("Playing video")
```

Now the caller can consistently use:

```
resource.open()
```

Common Mistake: Forcing Polymorphism Everywhere

Polymorphism is useful, but you should not create complicated class hierarchies just to use it.

If a simple function is enough:

```
def calculate_total(price, tax):
    return price + tax
```

there is no reason to create multiple classes.

Good OOP design is not about using every OOP feature.

It is about making the structure of the program easier to understand and maintain.

A Mental Model for Polymorphism

When designing a system, ask:

Step 1: What behavior is common?

Example:

`send()`

Step 2: Which different objects need that behavior?

`Email`
`SMS`
`Push`

Step 3: Should each object implement the behavior differently?

Yes.

Step 4: Can the calling code use the same method?

Yes:

`notification.send()`

That is a good candidate for polymorphism.

Polymorphism Design Pattern

A simple pattern looks like this:

```
class TypeA:
    def action(self):
        ...


class TypeB:
```

```
    def action(self):
        ...


class TypeC:
    def action(self):
        ...


def use_object(obj):
    obj.action()
```

Different objects:

```
use_object(TypeA())
use_object(TypeB())
use_object(TypeC())
```

The caller depends on the common behavior, not the concrete class.

When Should You Use Polymorphism?

Polymorphism is especially useful when:

- multiple objects perform the same conceptual action
- each object implements that action differently
- you want to add new types later
- you want to reduce large `if/elif` type checks
- you want reusable functions
- you are designing extensible systems
- different implementations should be interchangeable

Common examples include:

Payment methods
Notification systems
File processors
Database adapters
Storage systems
Authentication providers
UI components
Game characters
Reports
Export formats
haas.dev resources

When Not to Use It

Avoid unnecessary polymorphism when:

- there is only one implementation
- a simple function solves the problem
- the classes have no meaningful common behavior
- inheritance would make the design harder to understand
- you are creating abstractions before you actually need them

The goal is not:

> "Use polymorphism because OOP has it."

The goal is:

>

> "Use polymorphism when different objects genuinely need to provide the same behavior in different ways."

Mini Project: Notification System

Build a notification system using polymorphism.

Requirements

Create:

`EmailNotification`
`SMSNotification`
`PushNotification`

Each class should implement:

`send(message)`

Then create:

`send_notification(notification, message)`

The function should work with any notification object.

Expected usage

```
email = EmailNotification()
sms = SMSNotification()
push = PushNotification()
```

```
send_notification(email, "Welcome!")
send_notification(sms, "Your code is ready.")
send_notification(push, "You have a new message.")
```

The function should not need:

```
isinstance()
```

checks.

---

5 Minute Challenge

Create three classes:

```
PDF
Video
Quiz
```

Each must have:

```
open()
```

Make the output different for each class.

Then:

```
resources = [PDF(), Video(), Quiz()]

for resource in resources:
    resource.open()
```

Goal

Your loop should contain **no `if` or `elif` statements**.

Exercises

Exercise 1

Create:

`Dog`
`Cat`
`Cow`

Each should have:

`speak()`

Use one loop to call `speak()` on all objects.

Exercise 2

Create:

`CreditCard`
`PayPal`
`BankTransfer`

Each should have:

`pay(amount)`

Write:

`process_payment(payment, amount)`

that works with all three.

Exercise 3

Create:

`CSVExporter`
`JSONExporter`
`PDFExporter`

Each should have:

`export(data)`

Create one function that can use any exporter.

Exercise 4

Create three classes that do **not** inherit from the same parent but all implement:

`start()`

Use them through one function.

This exercise demonstrates Python's duck typing.

Mini Quiz

1. What does polymorphism mean?

A. One class can only have one object
B. Different objects can provide different behavior through a common interface
C. A class cannot inherit from another class
D. Variables can have multiple names

Answer: B

---

2. Is inheritance required for polymorphism in Python?

A. Always
B. Only with functions
C. No
D. Only with abstract classes

Answer: C

---

3. What does this demonstrate?

```
for resource in resources:
    resource.open()
```

when different resource types implement `open()` differently?

A. Encapsulation
B. Polymorphism
C. Recursion
D. Scope

Answer: B

---

4. What is duck typing?

A. Checking the exact class of every object
B. Using an object based on the behavior it provides
C. Creating a parent class
D. Converting objects into dictionaries

Answer: B

Interview Questions

Beginner

1. What is polymorphism in Python?

Polymorphism allows different objects to respond to the same method or interface in different ways.

2. Does Python support polymorphism without inheritance?

Yes. Python's duck typing allows objects to be used based on the behavior they provide.

3. What is method overriding?

It is when a child class provides its own implementation of an inherited method.

Intermediate

4. How is polymorphism related to inheritance?

Inheritance can provide a common interface and allow child classes to override behavior, but polymorphism itself does not require inheritance.

5. What is duck typing?

Duck typing means Python focuses on whether an object supports the required operation rather than requiring a specific class.

6. Why is polymorphism useful?

It allows common calling code to work with different implementations and makes systems easier to extend.

Practical

7. How would you design a payment system using polymorphism?

Create different payment classes that expose a common method such as:

`pay(amount)`

Each class implements payment differently.

8. Why can polymorphism reduce `if/elif` statements?

Instead of checking an object's type and deciding what to do externally, the object itself provides the appropriate implementation.

Polymorphism Cheat Sheet

```
Polymorphism
□
□□□ One interface
□   □
□□□ Multiple implementations
□   □
□□□ Same method call
□   □
□□□ Different behavior
```

Common pattern

```
class A:
    def action(self):
        print("A")


class B:
    def action(self):
        print("B")


objects = [A(), B()]
```

```
for obj in objects:
    obj.action()
```

With inheritance

```
class Parent:
    def action(self):
        print("Parent")


class Child(Parent):
    def action(self):
        print("Child")
```

Duck typing

```
def run(obj):
    obj.action()
```

Any object that provides:

```
action()
```

can potentially be passed to `run()`.

Key Takeaways

1. **Polymorphism means one interface can have different implementations.**
2. Different objects can respond to the same method in different ways.
3. Method overriding is a common way to achieve polymorphism through inheritance.
4. Python also supports polymorphism without inheritance.
5. Duck typing focuses on what an object can do rather than its exact type.

6. Built-in functions such as `len()` demonstrate polymorphic behavior.
7. Operators such as `+` can behave differently for different types.
8. Polymorphism can reduce unnecessary type-checking logic.
9. It makes systems easier to extend with new implementations.
10. Do not force polymorphism where a simple function or structure is clearer.

The core idea is:

```
Different objects
      
Same interface
      
Different behavior
```

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app