Method Overriding & Polymorphism

AiCE Software Engineering Foundations — Python Programming (Advanced)

Inheriting a parent's methods unchanged is useful, but often a child class needs to behave differently. This lesson covers method overriding — replacing an inherited method with a child-specific version — and polymorphism, the elegant consequence of overriding that lets generic code work correctly with many different specific object types.

1. Method Overriding

A child class can override a method inherited from the parent — replacing it with a different implementation. When you call the method on a child object, Python uses the child's version, not the parent's:

```
class Person:
    def __init__(self, name, age, email):
        self.name = name.strip().title()
        self.age = age
        self.email = email.strip().lower()

    def introduce(self):
        return f"Hi, I am {self.name}."

    def __str__(self):
        return f"Person: {self.name}"

class Student(Person):
    def __init__(self, name, age, email, course):
        super().__init__(name, age, email)
        self.course = course

    # Override introduce() with a student-specific version
    def introduce(self):
        return (f"Hi, I am {self.name}, a student enrolled in
{self.course}. "
                f"You can reach me at {self.email}.")
```

```
    # Override __str__
    def __str__(self):
        return f"Student: {self.name} ({self.course})"

class Instructor(Person):
    def __init__(self, name, age, email, specialisation):
        super().__init__(name, age, email)
        self.specialisation = specialisation

    # Override introduce() with an instructor-specific version
    def introduce(self):
        return (f"Hello, I am {self.name}, an instructor
specialising in "
                f"{self.specialisation}.")

    def __str__(self):
        return f"Instructor: {self.name}
({self.specialisation})"

# Create objects from each class
person = Person("Kwame Asante", 40, "kwame@aice.africa")
student = Student("Amara Diallo", 22, "amara@aice.africa",
"Python")
instructor = Instructor("Ngozi Adeyemi", 35,
"ngozi@aice.africa", "Data Engineering")

# Each calls its OWN version of introduce()
print(person.introduce())
print(student.introduce())
print(instructor.introduce())

print(person)
print(student)
print(instructor)
```

This produces:

```
Hi, I am Kwame Asante.
Hi, I am Amara Diallo, a student enrolled in Python. You can
reach me at amara@aice.africa.
Hello, I am Ngozi Adeyemi, an instructor specialising in Data
Engineering.
Person: Kwame Asante
Student: Amara Diallo (Python)
Instructor: Ngozi Adeyemi (Data Engineering)
```

Each class has its own version of introduce(). When you call the method on a person, student, or instructor, Python automatically uses the correct version for that object's type. This ability for different classes to implement the same method name differently — and for Python to use the correct one based on the object's type — is called polymorphism.

2. Polymorphism — One Interface, Many Behaviors

Polymorphism means 'many forms.' In Python OOP, it means that objects of different classes can be used interchangeably if they share a common interface — the same method names — even though their implementations differ.

The power of polymorphism is that code which works with a Person object can also work with Student and Instructor objects — because all three have the same methods. You can write a function that calls introduce() on any Person-derived object, and Python handles the rest:

```
def run_introductions(people):
    """Call introduce() on every object — works for any Person
subclass."""
    for person in people:
        print(f"  {person.introduce()}")

# A mixed list of Person, Student, and Instructor objects
attendees = [
    Person("Kwame Asante", 40, "kwame@aice.africa"),
    Student("Amara Diallo", 22, "amara@aice.africa",
"Python"),
```

```
    Instructor("Ngozi Adeyemi", 35, "ngozi@aice.africa", "Data
Engineering"),
    Student("Kofi Mensah", 19, "kofi@aice.africa",
"JavaScript"),
]

print("Meeting attendees:")
run_introductions(attendees)
```

The run_introductions() function does not need to know whether it is dealing with a Person, Student, or Instructor. It calls introduce() on each object and Python handles the correct version automatically. This is the elegance of polymorphism — generic code that works with specific objects.

> **Why Polymorphism Matters**
> *Without polymorphism, a function like run_introductions() would need a separate if statement checking each object's exact type before deciding what to call — a maintenance nightmare that grows worse every time a new type is added. Polymorphism removes that entirely: the function stays exactly the same no matter how many new Person subclasses you introduce later.*

Key Takeaways

Method overriding lets a child class redefine an inherited method entirely — Python always uses the child's version when the method is called on a child object. Polymorphism is the natural consequence: a function that calls a shared method name, like introduce(), works correctly on any mix of Person, Student, and Instructor objects, without ever needing to check which specific type it's handling. Generic code, specific behavior — that is the elegance polymorphism provides. The next lesson combines inheritance, overriding, and polymorphism into a complete, realistic class hierarchy modeling the entire AiCE platform, and covers how to check inheritance relationships directly, plus when to reach for composition instead of inheritance.